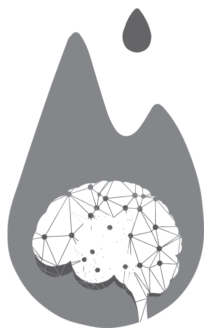
深度学习框架 PyTorch 快速开发与实战

邢梦来　王硕　孙洋洋　编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

LANZAROTE Caliente.com

内 容 简 介

深度学习已经成为人工智能炙手可热的技术，PyTorch 是一个较新的、容易上手的深度学习开源框架，目前已得到广泛应用。本书从 PyTorch 框架结构出发，通过案例主要介绍了线性回归、逻辑回归、前馈神经网络、卷积神经网络、循环神经网络、自编码模型、以及生成对抗网络。本书作为深度学习的入门教材，省略了大量的数学模型推导，适合深度学习初学者，人工智能领域的从业者，以及深度学习感兴趣的人阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

深度学习框架 PyTorch 快速开发与实战 / 邢梦来，王硕，孙洋洋编著. —北京：电子工业出版社，2018.8
ISBN 978-7-121-34564-7

Ⅰ. ①深… Ⅱ. ①邢… ②王… ③孙… Ⅲ. ①机器学习 Ⅳ. ①TP181

中国版本图书馆 CIP 数据核字（2018）第 135158 号

策划编辑：黄爱萍
责任编辑：张彦红
印　　刷：北京季蜂印刷有限公司
装　　订：北京季蜂印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：720×1000　1/16　印张：14.5　字数：232 千字
版　　次：2018 年 8 月第 1 版
印　　次：2018 年 8 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

前　　言

日常生活中，人工智能悄悄地影响着我们。

随着人工智能的技术的发展，现代几乎各种技术的发展都涉及了人工智能技术，可以说人工智能已经广泛应用到许多领域，其典型的应用包括：信息检索应用、推荐系统、语音识别、自然语言处理、图像识别、智能家居等。以人工智能在语音识别，语音合成上的结果看，2016 年 10 月份由微软美国研究院发布的一个语音识别的最新结果实现了错误率为 5.9%的新突破，这是第一次用人工智能技术取得了跟人类似的语音识别的错误率。

人工智能一直处于计算机技术的前沿，人工智能研究的理论和发现在很大程度上将决定计算机技术的发展方向。为了适应新一轮的科技发展，培养高端人才，人工智能进入国家发展战略。

2017 年 7 月，国务院印发《新一代人工智能发展规划》，其中提到，新一代人工智能发展分三步走的战略目标，到 2030 年使中国人工智能理论、技术与应用总体达到世界领先水平，成为世界主要人工智能创新中心。

为此，我们积极学习人工智能前沿知识，适应科技进步。

本书选用 Facebook 开源深度学习库 PyTorch 作为深度学习框架。常用的深度学习开源平台有 TensorFlow、Theano、Keras、Caffe 等。在 TensorFlow 的官网上，它被定义为一个用于机器智能的开源软件库，使用 TensorFlow 需要编写大量的代码，个人觉得不适合初学者。

Theano 是比较老牌和最稳定的库之一。由于 Theano 不支持多 GPU 扩展，在深度学习开源平台快速更新迭代的浪潮下，Theano 已然开始慢慢被遗忘了。

Keras 句法比较明晰，文档完善，使用非常简单轻松。Keras 强调极简主义，只需几行代码就能构建一个神经网络，适合新人学习。

Caffe 是老牌中的老牌框架。起初的时候它仅仅关注计算机视觉，但它具有非常好的通用性。Caffe 的缺点是它不够灵活，同时 Caffe 的文档非常贫乏。

张量是 PyTorch 的一个完美组件，和 NumPy 类似。将张量从 NumPy 转换至 PyTorch 非常容易。可以把它作为 NumPy 的替代品。PyTorch 这种框架可以获得 GPU 加速，以便快速进行数据预处理，或其他任务。PyTorch 同时也提供了变量，我们在构建神经网络的时候，在张量之上的封装，构建自己的计算图，并自动计算梯度。PyTorch 建立的是动态图，TensorFlow 建立的是静态图。PyTorch 更加符合一般的编程习惯，而不是像 TensorFlow 那样需要先定义计算图。

虽然开源平台众多，但更多的时候，我们考虑实现算法的简捷性，通常选择容易上手的，能快速实现算法的开源平台。为此，我们需要选择适合自己的深度学习开源平台，实现深度学习算法。

学习深度学习理论知识，了解人工智能行业发展动态，掌握前沿科学技术。利用 PyTorch 开源平台快速实现经典卷积神经网络、循环神经网络、自编码模型、对抗生成网络等模型。开启海绵模式，尽可能多学原理知识，掌握机器学习的基础理论知识，然后针对性地训练。通常从收集数据，预处理和清洗数据，到搭建模型，训练和调试模型，再到最后评估模型。逐渐培养出对于什么样的数据适合用什么类型的模型的判断能力，并增强实践能力。经过学习，逐渐从“小白”，慢慢到专业人士。有兴趣的读者欢迎加入本书交流群，一起交流学习。同时，本书所有案例的代码统一放在 QQ 群文件里，群号为 662443475，或在博文视点官网下载：www.broadview.com.cn。

致谢

感谢电子工业出版社的黄爱萍编辑，在选题策划和稿件整理方面做出的大量工作。

感谢极宽开源量化团队给予的技术支持。

在本书的创作中，特别感谢张建辉、刘笑俐、王丽颖、刘晓峰、刘婷、沈雨涵的协助，为他们的付出表示感谢。

邢梦来

2018 年 6 月

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/34564*



目　　录

第一部分　理论部分

第 1 章　深度学习简介 2
1.1　深度学习 2
1.2　神经网络的发展 6
1.3　深度学习的应用 7
1.4　常用的数学知识和机器学习算法 8
1.5　PyTorch 简介 11
1.5.1　PyTorch 介绍 11
1.5.2　使用 PyTorch 的公司 15
1.5.3　PyTorch API 16
1.5.4　为什么选择 Python 语言 16
1.5.5　Python 语言的特点 16
1.6　常用的机器学习、深度学习开源框架 17
1.7　其他常用的模块库 19
1.8　深度学习常用名词 20
第 2 章　PyTorch 环境安装 33
2.1　基于 Ubuntu 环境的安装 33
2.1.1　安装 Anaconda 35

2.1.2 设置国内镜像……36
2.2 Conda 命令安装 PyTorch……37
2.3 pip 命令安装 PyTorch……37
2.4 配置 CUDA……38
第 3 章 PyTorch 基础知识……40
3.1 张量……40
3.2 数学操作……43
3.3 数理统计……44
3.4 比较操作……45
第 4 章 简单案例入门……47
4.1 线性回归……47
4.2 逻辑回归……52
第 5 章 前馈神经网络……59
5.1 实现前馈神经网络……61
5.2 数据集……68
5.3 卷积层……72
5.4 Functional 函数……75
5.5 优化算法……82
5.6 自动求导机制……85
5.7 保存和加载模型……87
5.8 GPU 加速运算……87
第 6 章 PyTorch 可视化工具……89
6.1 Visdom 介绍……89
6.2 Visdom 基本概念……90
6.2.1 Panes（窗格）……90
6.2.2 Environments（环境）……90
6.2.3 State（状态）……91
6.3 安装 Visdom……91
6.4 可视化接口……91

6.4.1 Python 函数属性提取技巧 92
6.4.2 vis.text 93
6.4.3 vis.image 93
6.4.4 vis.scatter 94
6.4.5 vis.line 95
6.4.6 vis.stem 97
6.4.7 vis.heatmap 97
6.4.8 vis.bar 99
6.4.9 vis.histogram 101
6.4.10 vis.boxplot 102
6.4.11 vis.surf 103
6.4.12 vis.contour 104
6.4.13 vis.mesh 106
6.4.14 vis.svg 107

第二部分　实战部分

第 7 章　卷积神经网络 110
7.1 卷积层 112
7.2 池化层 114
7.3 经典的卷积神经网络 115
7.3.1 LeNet-5 神经网络结构 115
7.3.2 ImageNet-2010 网络结构 117
7.3.3 VGGNet 网络结构 122
7.3.4 GoodLeNet 网络结构 124
7.3.5 ResNet 网络结构 126
7.4 卷积神经网络案例 129
7.5 深度残差模型案例 138
第 8 章　循环神经网络简介 145
8.1 循环神经网络模型结构 146
8.2 不同类型的 RNN 147
8.3 LSTM 结构具体解析 151

8.4 LSTM 的变体 153
8.5 循环神经网络实现 156
8.5.1 循环神经网络案例 156
8.5.2 双向 RNN 案例 160
第 9 章 自编码模型 164
第 10 章 对抗生成网络 172
10.1 DCGAN 原理 175
10.2 GAN 对抗生成网络实例 180
第 11 章 Seq2seq 自然语言处理 186
11.1 Seq2seq 自然语言处理简介 186
11.2 Seq2seq 自然语言处理案例 188
第 12 章 利用 PyTorch 实现量化交易 204
12.1 线性回归预测股价 205
12.2 前馈神经网络预测股价 209
12.3 递归神经网络预测股价 214

1

第一部分

理论部分

第 1 章　深度学习简介
第 2 章　PyTorch 环境安装
第 3 章　PyTorch 基础知识
第 4 章　简单案例入门
第 5 章　前馈神经网络
第 6 章　PyTorch 可视化工具

1 第 1 章

深度学习简介

在北京时间 2016 年 3 月 15 日的下午，谷歌 DeepMind 团队开发的围棋深度学习系统 AlphaGo 以总比分 4∶1 战胜了韩国棋手李世石，成为第一个在 19×19 棋盘上战胜人类围棋冠军的智能系统。2017 年 10 月 19 日，DeepMind 团队重磅发布 AlphaGo Zero。相比上一代 AlphaGo，该版本的 AlphaGo实现了在AI发展中非常有意义的一步——“无师自通”。在AlphaGo的核心组成部分中，估值网络（Value Network）和走棋网络（Policy Network）都使用了深度学习的技术。AlphaGo 战胜李世石的新闻成功地把深度学习的概念从学术界推向了大众，并点燃了大众对于人工智能的巨大热情。AlphaGo Zero 的伟大之处是第一次让机器可以不通过任何棋谱，不通过任何人类的经验，在只告诉规则的前提下，成为一个围棋高手，这种无师自通的学习模式在 AI 整个发展历史上是非常有意义的。

1.1　深度学习

深度学习的概念由 Hinton 等人于 2006 年提出。深度学习的概念源于人工神经网络的研究。含多隐层的多层感知器就是一种深度学习结构。深度学习通过组合低层特征形成更加抽象的高层次表示属性类别或特征，以发现数据的分布式特征表示。

深度学习在完成一些难度极高的任务中展现了惊人的功力，如从图片中识别物体、语言理解，棋盘类游戏等。我们举一个例子来具体说明什么是深度学习。

我们都知道机器学习的目的是在没有特定编程的情况下，希望系统去回答某个问题。比如：明天北京会下雨吗？这类问题可以翻译成以下形式：对于给定的输入 X，正确的输出 Y 是什么？输入的是北京天气的信息，输出则为会下雨或者不会下雨。

传统的机器学习需要我们首先定义一系列程序寻找的特征。比如，图 1.1 里的猫有两个眼睛一个鼻子，四条腿，以及它毛茸茸的毛。我们给程序提供大量的例子，但是对于每个例子，都不会向程序展示全局图片，而是展示预先设定的某特征的变量，然后告诉程序哪个才是猫。经过训练之后，程序就能明白如果它没有毛茸茸的毛，它可能不是猫。而深度学习却可以很好地解决这个问题。



图 1.1　使用深度学习识别图片中的猫

深度学习模型中的卷积神经网络模型可以用来解决这个问题，通过卷积神经网络我们来讲讲深度学习模型是如何识别图片里的猫的。图 1.2 是简单的单层神经网络模型。

第一步，我们准备好要识别的图片，假设这张图片的尺寸为 28×28 像素，然后把图片中的像素传输给卷积神经网络模型。第一层神经网络将扫描图片，以 5×5 的色块为单位，去寻找一些基本的特征，并提取特征，形成特征地图。

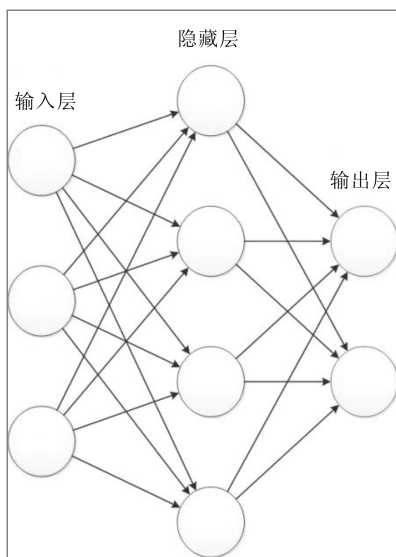


图 1.2　单层神经网络模型

第二层将把第一层产生的特征地图，继续按照第一次的方式扫描，这样一层又一层，直到有一层获取了足够的信息可以判断这是一只猫。我们可以看到卷积神经网络很神奇地筛选出猫的特征来，而采用卷积神经网络我们只需要设定模型结构，让程序从训练数据中自己归纳学习，模型会自动采集重要的模型特征，从而判断出是猫或者不是猫。

我们从深度学习的深度和学习两方面进行讲解。

深度学习的前身是人工神经网络。我们说的神经网络一般就是指人工神经网络。最简单的神经网络由输入层、隐藏层和输出层组成。输入训练数据为输入层，输出计算结果为输出层，隐藏层使输入数据传播到输出层，从而把神经网络形成网络结构。传统神经网络的每一层有大量的节点组成，也叫神经元。每层内的节点相互独立、互不干扰。层与层之间的节点相互连接，深度学习就是增加多层网络结构，利用现有的数据，来对未知的数据做预测分类。

深度学习的模型是如何学习的呢？我们也许都有过这样的经历，做过一道数学题并且知道如何求解，下次再遇到类似的题目就可以很轻松地解决。我们在学习的过程中举一反三、触类旁通，逐渐地拥有解决问题的能

力。但是我们都知道机器没有大脑思维，如何才能让机器自己学习并拥有这样的能力呢？这也是人工智能发展的方向。现在我们会问深度学习是否有这样的能力呢？

我们给设定好的机器模型输入大量的数据，不断地训练，最终得到我们想要的结果。这些过程就像我们平时做大量练习题，不断地思考并且训练如何能正确地解答题目，到最后获得正确的解决问题、得出答案的能力。

我们再从监督学习和无监督学习两方面进行讲解。

机器学习方法有监督学习与无监督学习之分，和机器学习方法一样，深度机器学习方法也有监督学习与无监督学习之分。不同的学习框架下建立的学习模型也不同。

有监督学习：利用一组已知类别的样本调整分类器的参数，使其达到所要求性能的过程，也称为监督训练或有教师学习。对具有概念标记（分类）的训练样本进行学习，以尽可能对训练样本集外的数据进行标记（分类）预测。这里，所有的标记（分类）是已知的。因此，训练样本的歧义性低。

无监督学习：对没有概念标记（分类）的训练样本进行学习，以发现训练样本集中的结构性知识。这里，所有的标记（分类）是未知的。因此，训练样本的歧义性高。聚类就是典型的无监督学习。

半监督学习（Semi-supervised Learning）是模式识别和机器学习领域研究的重点问题，是监督学习与无监督学习相结合的一种学习方法。它主要考虑如何利用少量的标注样本和大量的未标注样本进行训练和分类的问题。半监督学习对于减少标注代价，提高机器学习性能具有非常重大的意义。

虽然 AlphaGo 战胜李世石将人工智能推向了一个新的高度，但是 AlphaGo 能够解决的仅仅是在一个特定环境中定义好的问题，要将人工智能系统真正的应用到开放环境，还需要研究人员更多的努力。这也将是 AI 未来发展的方向。

1.2 神经网络的发展

2017 年，人工智能发展火热。机器学习和大数据领域取得突破性的进展，作为人工智能一个重要分支的深度学习，也正受到大家越来越多的关注。经历了几年的高速发展，深度学习在工业界和学术界备受追捧，百度、阿里巴巴、腾讯也将人工智能方面的人才培养作为战略重心，同时吹响了人才抢夺的号角。斯坦福大学教授、计算机视觉领域领军人物李飞飞（Feifei Li）于 2016 年加入谷歌；卡内基梅隆大学教授、机器学习领域顶级人物 Alex Smola 于 2016 年加入亚马逊。深度学习界泰斗吴恩达（Andrew Ng）宣布退出百度，成立了自己的人工智能公司，2017 年 8 月 15 日，吴恩达向美国证券交易委员会（SEC）注册了一支 1.5 亿美元的风险投资基金，专门投资人工智能领域。

除吴恩达外，在过去的几个月中，谷歌也公布了一项专注人工智能领域的风险投资基金——Gradient Ventures，它将为人工智能领域的初创公司提供科研资金和技术指导。Dropbox 前创始人和腾讯也一起创立了 1.36 亿美元的 Basis Set Ventures，同样专注人工智能领域。此外，Element.AI 筹集了 1.02 亿美元；微软创立了自己的人工智能基金；丰田也筹备了 1 亿美元用于人工智能投资。

深度学习为什么会这么火？我们也许可以从生活中找到答案。比如阿里巴巴在杭州的无人超市，百度正在研发的无人驾驶的汽车，人脸识别软件等，这些都是深度学习在生活中的应用。也许很多人都觉得奇怪，看似没有任何情感的机器如何能完成一些复杂的操作呢？

下面开始介绍深度学习的发展历程，也许会找到相关答案。

深度学习属于机器学习一个分支，同时机器学习又是人工智能的一个子集。什么叫作人工智能呢？在 20 世纪 40 年代和 50 年代，来自不同领域的一批科学家开始探讨制造人工大脑的可能性。1956 年，人工智能被确立为一门学科。人工智能是研究、开发用于模拟、延伸和扩展人的智能的理论、方法、技术及应用系统的一门新的技术科学。相信大家都听说过“图灵测试”吧？图灵测试一词来源于计算机科学和密码学的先驱阿兰·麦席森·图灵写于 1950 年的一篇论文《计算机器与智能》。阿兰·麦席森·图

灵 1950 年设计出这个测试，其内容是，如果电脑能在 5 分钟内回答由人类测试者提出的一系列问题，且其超过 30%的回答让测试者误认为是人类所答，则电脑通过测试。如果电脑通过测试，则说明智能程度相对较高。为此，科学家正在研究如何让机器像人类一样思考。

什么叫作机器学习呢？机器学习（Machine Learning，ML）是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科。它专门研究计算机怎样模拟或实现人类的学习行为，以获取新的知识或技能，重新组织已有的知识结构使之不断改善自身的性能。它是人工智能的核心，是使计算机具有智能的根本途径，其应用遍及人工智能的各个领域，它主要使用归纳、综合而不是演绎的方法。

1.3　深度学习的应用

2017 年 3 月 13 日，AlphaGo 与李世石的第四场对决结束，在连输 3 场之后，李世石终于扳回一局。但 3：1 的比赛结果已说明人工智能的强大，这也是谷歌对深度学习、人工智能的成功营销。AlphaGo 在短短几个月实现性能的大幅提升，用五个月走完了 IBM“深蓝”4 年的路，体现了当前人工智能系统学习速度之快。

2017 年 10 月 30 日，*MIT Technology Review*、*Slate*、*Quartz*、*Gear of Biz* 等美国媒体发表文章称，用不了多长时间，AlphaGo 将不再是地球上最好的棋手。之后新式高超的人工智能程序版本 AlphaGo Zero 出现，它堪称怪物。它从零开始，面对的只是一张空白的棋盘和游戏规则。它无师自通，仅仅通过自学使自己的游戏技能得以提高。但是它从来都不仅仅关乎棋盘游戏，未来它将会在更多领域发挥作用。

近年来，深度学习已经在图像识别、语音识别等领域获得了一些应用。目前深度学习技术应用最多的还是视觉领域，即对图像和视频的分析。在图像分析方面，比如人们熟悉的人脸识别、文字识别和大规模图像分类等。深度学习大幅提升了复杂任务分类的准确率，使得图像识别、语音识别以及语义理解准确率大幅提升。在语音识别等领域应用有 iPhone 的语音助理 Siri、百度的度秘、科大讯飞的“灵犀”、微软的小冰等。

深度学习有很多应用场景，只要涉及目标检测、目标识别的地方，理论上都可以应用深度学习来解决。就像百度前首席科学家吴恩达在一些报告中提到的，深度学习可以取代现有的很多特征提取、目标检测技术。

人脸识别是众多的人工智能技术之一。日常生活中我们可以轻而易举地通过看人的面孔识别其身份。对于机器而言，需要把传入的图片转换成数字，通过算法一步一步来提取图片里面的像素特征，来达到识别人脸的特征，从而快速做出决策。在深度学习的帮助下，人脸识别算法已经达到它的鼎盛时期了，识别准确率甚至超过了人眼。

其实对几乎所有人工智能问题，如何通过更高层次的抽象来理解输入，从而更快速地做出决策都是解决问题的关键所在。深度学习是最近一段时间引领新一波人工智能浪潮的核心技术，正是看到深度学习技术如此巨大的潜力，国际互联网巨头 Google、Facebook、Microsoft 纷纷抢先布局，争夺技术人才。

1.4 常用的数学知识和机器学习算法

概率论和统计学：机器学习和统计学的关系。实际上，有人最近将机器学习定义为“在 Mac 上做统计”。机器学习所需的一些基本统计和概率理论主要有：组合学、概率规则和公理、贝叶斯定理、随机变量、方差和期望、条件和联合分布、标准分布（伯努利分布、二项式分布、多项式分布、均匀分布和高斯分布等）、动差生成函数、最大似然估计、先验和后验、最大后验估计和抽样方法。

多元微积分：一些必要的内容包括微积分、偏导数、向量值函数、方向梯度、Hessian、Jacobian、Laplacian 和 Lagragian 分布。所以，千万别被机器学习中的数学所吓倒而不知道该如何下手。只要有上述几门课的基础，你完全可以看懂很大一部分机器学习算法。入门之后，我们需要矩阵分析、优化设计、离散数学、算法和优化理论。算法和优化理论对我们理解机器学习算法的计算效率和可拓展性，以及怎么利用数据中的稀疏性很重要。需要的知识主要包括：数据结构（二叉树、散列、堆、堆栈等）、动态规划、随机和次线性算法、图论、梯度/随机下降和原始-对偶方法。其他还包括：

复变函数（集合和序列、拓扑结构、度量空间、单值和连续函数、极限等）、信息论（熵、信息增益）、函数空间和流形。进一步学习数学方面的知识，增加自己的视野，通过数学知识寻找更多解决问题的方法。正如 Daniel Jeffries 在其系列文章 *Learning AI if You Suck at Math* 中所说，数学可以帮你更加清楚地理解机器学习的深层含义，但是只要你具备了一些数学基础，那么你马上就可以开始运用。

掌握了最基础的数学知识之后，我们开始学习经典的机器学习理论与基本算法。

1．决策树（Decision Tree）算法

决策树是在已知各种情况发生概率的基础上，通过构成决策树来求取净现值的期望值大于或等于零的概率，从而评价项目风险，判断其可行性的决策分析方法，是直观运用概率分析的一种图解法。由于这种决策分支画成图形很像一棵树的枝干，故称决策树。在机器学习中，决策树是一个预测模型，它代表的是对象属性与对象值之间的一种映射关系。

2．K-Means 算法（The k-means algorithm）

K-Means 算法算法是一个聚类算法，把 n 的对象根据他们的属性分为 k 个分割，$k < n$。它与处理混合正态分布的最大期望算法很相似，因为它们都试图找到数据中自然聚类的中心。它假设对象属性来自空间向量，并且目标是使各个群组内部的均方误差总和最小。

3．支持向量机（Support Vector Machine，SVM）

支持向量机（Support Vector Machine，SVM），简称 SV 机（论文中一般简称 SVM）。它是一种监督式学习的方法，它广泛应用于统计分类以及回归分析中。支持向量机将向量映射到一个更高维的空间里，在这个空间里建立了一个最大间隔的超平面。在分开数据的超平面的两边建有两个互相平行的超平面。分隔超平面使两个平行超平面的距离最大化，并假定平行超平面间的距离或差距越大，分类器的总误差越小。

4．The Apriori algorithm

Apriori 算法是一种挖掘关联规则的频繁项集算法，其核心思想是通过

候选集生成和情节的向下封闭检测两个阶段来挖掘频繁项集。该算法已经被广泛地应用到商业、网络安全等领域。

5．最大期望（EM）算法

最大期望算法（Expectation-maximization algorithm，又译期望最大化算法）在统计中被用于寻找，依赖于不可观察的隐性变量的概率模型中，参数的最大似然估计。

在统计计算中，最大期望（EM）算法是在概率（Probabilistic）模型中寻找参数最大似然估计或者最大后验估计的算法，其中概率模型依赖于无法观测的隐藏变量（Latent Variable）。最大期望经常用在机器学习和计算机视觉的数据聚类（Data Clustering）领域。最大期望算法经过两个步骤交替进行计算，第一步是计算期望（E），利用对隐藏变量的现有估计值，计算其最大似然估计值；第二步是最大化（M），最大化在 E 上求得的最大似然值来计算参数的值。M 上找到的参数估计值被用于下一个 E 计算中，这个过程不断交替进行。

6．PageRank 网页排名

PageRank 是 Google 排名运算法则（排名公式）的一部分，是 Google 用来标识网页的等级/重要性的一种方法，是 Google 用来衡量一个网站好坏的唯一标准。在融合了诸如 Title 标识和 Keywords 标识等所有因素之后，Google 通过 PageRank 来调整结果，使那些更具“等级/重要性”的网页在搜索结果中获得排名提升，从而提高搜索结果的相关性和质量。

7．AdaBoost

AdaBoost 是一种迭代算法，其核心思想是针对同一个训练，集训练不同的分类器（弱分类器），然后把这些弱分类器集合起来，构成一个更强的最终分类器（强分类器）。

8．*k*-NN

k 最近邻(*k*-Nearest Neighbor，*k*-NN)分类算法，该方法的思路是：如果一个样本在特征空间中的 *k* 个最相似（即特征空间中最邻近）的样本中的大多数属于某一个类别，则该样本也属于这个类别。是一个理论上比较成

熟的方法，也是最简单的机器学习算法之一。

9．Naive Bayes 朴素贝叶斯

贝叶斯分类器的分类原理是通过某对象的先验概率，利用贝叶斯公式计算出其后验概率，即该对象属于某一类的概率，选择具有最大后验概率的类作为该对象所属的类。朴素贝叶斯模型发源于古典数学理论，有着坚实的数学基础，以及稳定的分类效率。

当拥有了最基础的知识和学会常用的机器学习算法后，我们就可以去尝试做一些深度学习的模型来试一试模型的效果。带着问题出发，逐渐形成自己对模型的理解与感悟，尝试着修改一些经典的深度学习模型的算法，加深对深度学习模型的理解与应用。

1.5　PyTorch简介

1.5.1　PyTorch 介绍

2017 年 1 月份，Facebook 开源了 PyTorch。PyTorch 由 Adam Paszke、Sam Gross 与 Soumith Chintala 等人牵头开发，其成员来自 Facebook FAIR 和其他多家实验室。PyTorch 的前身是 Torch。Torch 是一个科学计算框架，支持机器学习算法，易用而且提供高效的算法实现，这得益于 LuaJIT 和一个底层的 C 实现。由于 Torch 由 Lua 语言编写，Torch 在神经网络方面一直表现得很优异，但是 Lua 语言不怎么流行，从而开发者把 Torch 移植到 Python 语言中，形成了 PyTorch。所以，也可以说 PyTorch 是 Torch 在 Python 上的移植。其图标如图 1.3 所示。



图 1.3　PyTorch 图标

目前 PyTorch 仅支持 Linux，Mac 平台的操作系统，截至 2017 年 12 月 PyTorch 最新的版本是 PyTorch 0.2。其应用版本环境如图 1.4 所示。

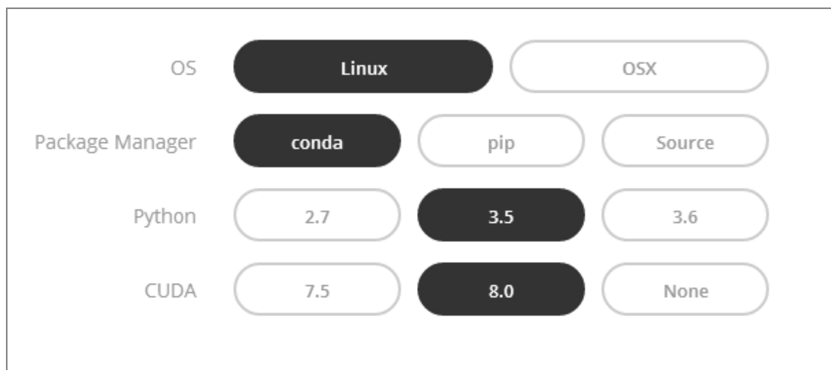


图 1.4　PyTorch 应用版本环境

PyTorch 的官网地址：http://pytorch.org/，其官网界面如图 1.5 所示。

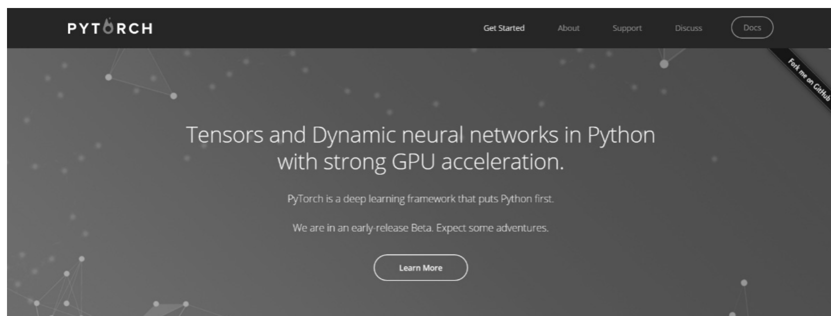


图 1.5　PyTorch 官网界面

Python 作为功能十分强大的高级语言，擅长处理数据分析，拥有众多模块包，是众多数据分析爱好者的选择。PyTorch 支持动态图的创建。现在的深度学习平台在定义模型的时候主要用两种方式：静态图模型（Static computation graph）和动态图模型（Dynamic computation graph）。静态图定义的缺陷是在处理数据前必须定义好完整的一套模型，能够处理所有的边际情况。动态图模型能够非常自由地定义模型。使用和重放 Tape recorder 可以零延迟或零成本地任意改变你的网络的行为。动态图模型作为 NumPy 的替代者，使用强大的 GPU，支持 GPU 的 Tensor 库，可以极大地加速计算。PyTorch 的设计思路简单实用，PyTorch 将会直接指向代码定义的确切位置，节省开发者寻找 Bug 的时间。同时根据代码简介，可快速实现神经网络构建，还有 Lua 的社区支持，为 PyTorch 提供各种技术支持和交流。

如果你使用 NumPy，那么你已经在使用 Tensors（也就是 ndarray）。PyTorch 提供的 Tensors 支持 CPU 或 GPU，并为大量的计算提供加速。

Autograd 实现是非常重要的一个功能。变量和功能是相互关联的，可以建立一个无环图，编码一个完整的历史的计算，并且每个变量都有一个 grad_fn。

PyTorch 提供的功能有强大的 N 维数组，提供大量索引、切片和置换的程序，通过 LuaJIT 实现神奇的 C 接口、线性算术程序、神经网络以及以能源为基础的模型。在 2016 年，谷歌开源了数值优化程序 TensorFlow，用 Tensor 的形式，在静态的神经网络上大大提高了运行效率，相比于 PyTorch，它提供了动态的神经网络计算图模块，方便用户随时改变神经网络的结构，同时也提供 GPU 加速，在不影响计算的情况下，实现快速搭建神经网络。许多大公司为了提高科研效率，选择简单而且方便的 Python 作为开发工具，比如说谷歌，Facebook。并且 TensorFlow 和 PyTorch 都已经兼容 Python 3.5，可以实现快速搭建自己的神经网络。但是 Torch 缺乏 TensorFlow 的分布式应用程序管理框架，缺乏多种编程语言的 API，限制了开发人员，影响 PyTorch 在深度学习领域中的地位。

PyTorch 支持卷积神经网络、循环神经网络以及长短期记忆网络。同时 PyTorch 提供了大量的图片数据方便用户进行实验，如图 1.6 所示。

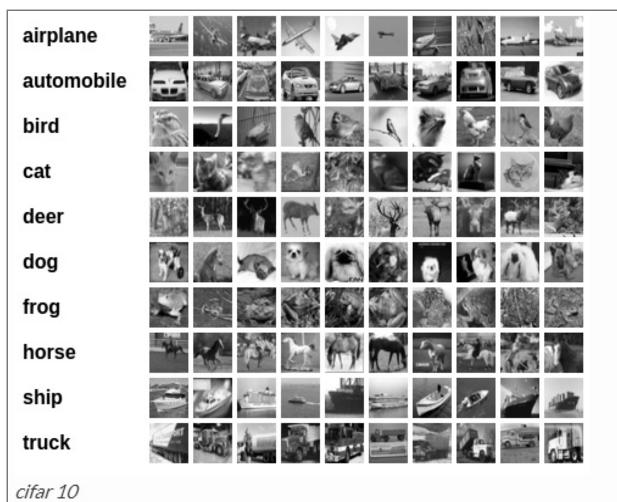


图 1.6　cifar10 内置的图像数据

图 1.7 是用 PyTorch 做人脸识别的图像处理。

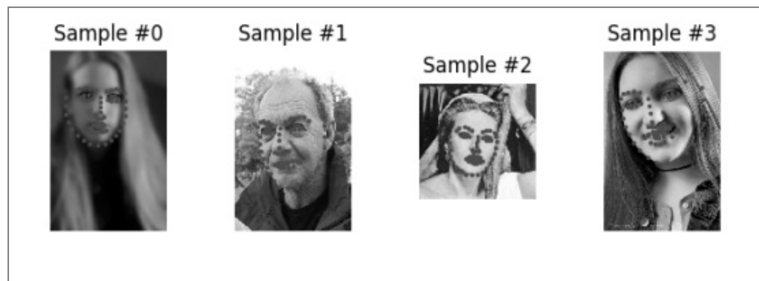


图 1.7　人脸识别的图像处理

我们来看一下 PyTorch 的模型运作流程图（如图 1.8 所示）：从输入 input 到输出 output 很简洁，也很直观。

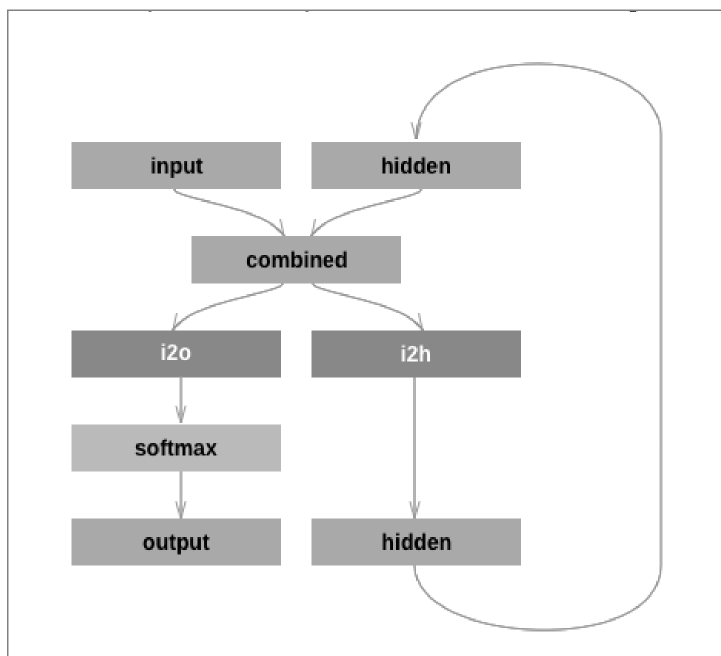


图 1.8　PyTorch 的网络模型运作流程图

图 1.9 是 PyTorch 与 Torch 训练 RNN 模型按流程的对比图，PyTorch 相比于 Torch 而言更加简洁，如果你想创建一个递归网络，只需多次使用相同的线性层，无须考虑共享权重。

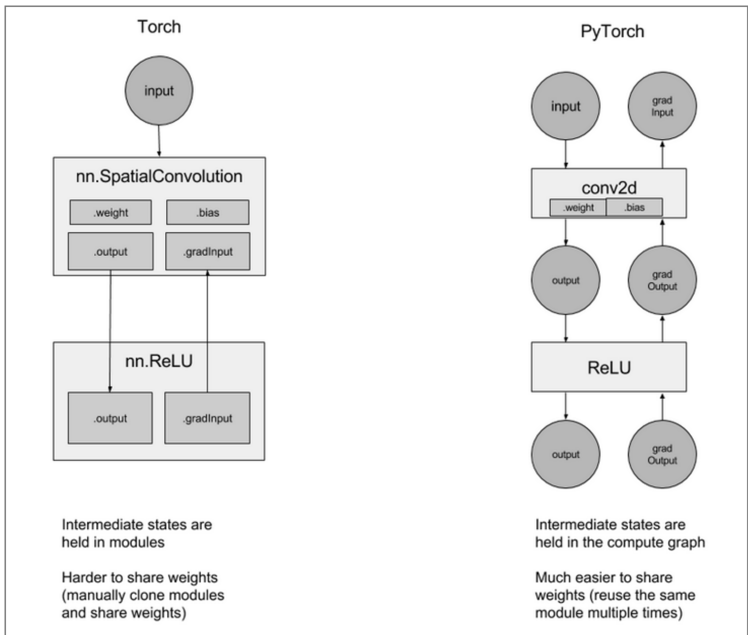


图 1.9　PyTorch 与 Torch 训练 RNN 模型按流程对比图

1.5.2　使用 PyTorch 的公司

由于 PyTorch 符合直觉、好理解、易用。越来越受广大程序员的喜爱。下面我们来看看使用 PyTorch 的公司吧。

目前除了 Facebook 公司使用 PyTorch 之外，还有 NVIDIA 公司等著名的企业使用。图 1.10 是使用 PyTorch 公司的截图。

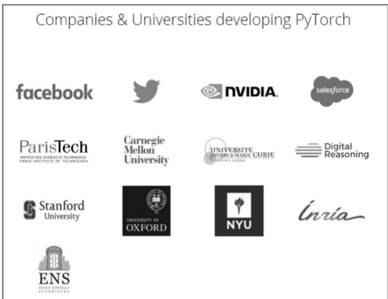


图 1.10　使用 PyTorch 的公司

1.5.3 PyTorch API

查询 PyTorch 的 API，以及对外提供的 API 的详细使用方法，读者可以自己通过链接查询：http://pytorch.org/docs/0.3.0/。在此不再叙述。

1.5.4 为什么选择 Python 语言

Python，是一种面向对象的解释性的计算机程序设计语言，也是一种功能强大而完善的通用型语言，已经具有十多年的发展历史，成熟且稳定。Python 具有脚本语言中最丰富和强大的类库，足以支持绝大多数日常应用。

1.5.5 Python 语言的特点

Python 是一种十分精彩又强大的语言。它合理地结合了高性能与编写程序简单有趣的特色。Python 是一种代表简单主义思想的语言。就如同你即将看到的一样，Python 极其容易上手。Python 有极其简单的语法。Python 是 FLOSS（自由/开放源码软件）之一。简单地说，你可以自由地发布这个软件的拷贝文件、阅读它的源代码、对它做改动、把它的一部分用于新的自由软件中。当你用 Python 语言编写程序的时候，你无须考虑诸如如何管理你的程序使用的内存一类的底层细节。

由于它的开源本质，Python 已经被移植在许多平台上。所有 Python 程序无须修改就可以在下述任何平台上面运行。这些平台包括 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE，甚至还有 PocketPC 和 Symbian！Python，即支持面向过程的编程也支持面向对象的编程。在“面向过程”的语言中，程序是由过程或仅仅是可重用代码的函数构建起来的。在“面向对象”的语言中，程序是由数据和功能组合而成的对象构建起来的。与其他主要的语言如 C++和 Java 相比，Python 以一种非常强大又简单的方式实现面向对象编程。同时你可以把 Python 嵌入你的 C/C++程序，从

而向你的程序用户提供脚本功能。Python 标准库确实很庞大。它可以帮助你处理各种工作，包括正则表达式、线程、数据库、网页浏览器、GUI（图形用户界面）等。

1.6　常用的机器学习、深度学习开源框架

有了上述的基础后，想要深入学习，最好的办法就是边学习边实践。有一句古话说得好，实践出真知。带着问题进行训练模型，不仅可以增加自己对问题思考的深度，举一反三，还可以让自己迅速掌握模型的精髓，懂得什么时候选择什么样的模型效果更好，提高训练数据的效果。研究经典的论文，体会建模的思想，学会处理各种复杂数据，进一步优化模型效果。

有了理论依据，是时候动手写代码测试模型的效果了。选择一个开源的深度学习框架，可以让自己不重复造轮子，减少因编程带来的训练模型的困难。下面开始介绍目前比较流行的深度学习模块库。

1．PyTorch

2017 年年初，Facebook 在机器学习和科学计算工具 Torch 的基础上，针对 Python 语言发布了一个全新的机器学习工具包 PyTorch。Python 作为功能十分强大的高级语言，擅长处理数据分析，拥有众多模块包，是众多数据分析爱好者的选择。PyTorch 支持动态图的创建。作为 NumPy 的替代，以便使用强大的 GPU，支持 GPU 的 Tensor 库，可以极大地加速计算。PyTorch 的设计思路简单实用，PyTorch 将会直接指向代码定义的确切位置，节省开发者寻找 BUG 的时间。同时代码简介，可快速实现神经网络构建，还有 Lua 的社区支持，为 PyTorch 提供各种技术支持和交流。

2．TensorFlow

TensorFlow 是谷歌的第二代机器学习系统，2015 年 11 月 9 日，谷歌发布人工智能系统 TensorFlow 并宣布开源。TensorFlow 内建深度学习的扩展支持，任何能够用计算流图形来表达的计算，都可以使用 TensorFlow。任何基于梯度的机器学习算法都能够受益于 TensorFlow 的自动分化

（Auto-differentiation）。TensorFlow 是一个采用数据流图（Data flow graphs），用于数值计算的开源软件库。节点（Nodes）在图中表示数学操作，图中的线（Edges）则表示在节点间相互联系的多维数据数组，即张量（Tensor）。它灵活的架构让你可以在多种平台上展开计算，例如台式计算机中的一个或多个 CPU（或 GPU）、服务器、移动设备等。TensorFlow 具有高度的灵活性，真正的可移植性，支持多种语言，同时具有性能最优化的特点，紧密地将科研和产品联系在一起。

3．Caffe

Caffe 的全称是 Convolutional Architecture for Fast Feature Embedding，它是一个清晰、高效的深度学习框架，它是开源的，核心语言是 C++，它支持命令行、Python 和 Matlab 接口，它既可以在 CPU 上运行，也可以在 GPU 上运行。它的 License 是 BSD 2-Clause。Caffe 从一开始就设计得尽可能地模块化，允许对新数据格式、网络层和损失函数进行扩展。Caffe 的模型（Model）定义是用 Protocol Buffer 语言写进配置文件的。以任意有向无环图的形式，Caffe 支持网络架构。Caffe 会根据网络的需要来正确占用内存。通过一个函数调用，实现 CPU 和 GPU 之间的切换。在 Caffe 中，每一个单一的模块都对应一个测试。同时提供 Python 和 Matlab 接口。针对视觉项目，Caffe 提供了一些参考模型，这些模型仅应用在学术和非商业领域使用。

4．Scikit-Learn

Scikit-Learn 是基于 Python 的机器学习模块，基于 BSD 开源许可证。这个项目最早由 David Cournapeau 在 2007 年发起的，目前也是由社区自愿者进行维护。Scikit-Learn 的基本功能主要被分为六个部分，分类、回归、聚类、数据降维、模型选择，数据预处理，对于具体的机器学习问题，通常可以分为三个步骤，数据准备与预处理，模型选择与训练，模型验证与参数调优，Scikit-Learn 支持多种格式的数据，包括经典的 IRIS 数据，LibSVM 格式数据等。

1.7　其他常用的模块库

1．Matplotlib

Matplotlib 是基于 Python 语言的开源项目，旨在为 Python 提供一个数据绘图包。Matplotlib 项目是由 John D. Hunter 发起的。John D. Hunter 由于癌症于 2012 年去世，但他为社区作出的无比贡献将永远留存。Matplotlib 是一个 Python 的 2D 绘图库，它以各种硬拷贝格式和跨平台的交互式环境生成出版质量级别的图形。通过 Matplotlib，开发者仅需要几行代码，便可以生成绘图、直方图、功率谱、条形图、散点图等。Matplotlib 的对象体系严谨而有趣，为使用者提供了巨大的发挥空间。用户在熟悉了核心对象之后，可以轻易地定制图像。Matplotlib 的对象体系也是计算机图形学的一个优秀范例。即使你不是 Python 程序员，也可以从文中了解一些通用的图形绘制原则。

2．NumPy

NumPy 系统是 Python 的一种开源的数值计算扩展。这种工具可用来存储和处理大型矩阵，比 Python 自身的嵌套列表结构要高效得多。NumPy 包含很多实用的数学函数，涵盖线性代数运算、傅立叶变换和随机生成等功能。

3．Pandas

Pandas 是 Python 的一个数据分析包，最初由 AQR Capital Management 于 2008 年 4 月开发，并于 2009 年年底开源出来，目前由专注于 Python 数据包开发的 PyData 开发团队继续开发和维护。Pandas 最初被作为金融数据分析工具而开发出来，因此，Pandas 为时间序列分析提供了很好的支持。Pandas 是基于 NumPy 的一种工具，该工具是为了解决数据分析任务而创建的。Pandas 纳入了大量库和一些标准的数据模型，提供了高效地操作大型数据集所需的工具。Pandas 提供了大量能使我们快速便捷地处理数据的函数和方法。

1.8 深度学习常用名词

几年之前，深度学习还是机器学习中一个不太受人关注的领域。随着最近神经网络和大数据概念的出现，很多复杂任务的实现已经成为可能。

目前，深度学习已经被应用到很多的领域当中，例如语音识别、图像识别、在一个数据集当中寻找模式、照片中的事物分类、字符文本生成、自动驾驶汽车等等。因此，了解深度学习及其概念是非常重要的。

1. 人工神经网络（ANNs）

人工神经网络（Artificial Neural Networks，简写为 ANNs）是一种模仿动物神经网络行为特征，进行分布式并行信息处理的算法数学模型。这种网络依靠系统的复杂程度，通过调整内部大量节点之间相互连接的关系，从而达到处理信息的目的，并具有自学习和自适应的能力。

2. 生物神经元

神经系统的基本结构和功能单位是神经细胞，即神经元（Neurons）。无脊椎动物和脊椎动物的神经元形态相似，都是由细胞体和从细胞延伸的突起所组成。1943 年，心理学家 McCulloch 和数学家 Pitts 参考了生物神经元的结构，发表了抽象的神经元模型 MP。神经元模型是一个包含输入、输出与计算功能的模型。输入可以类比为神经元的树突，而输出可以类比为神经元的轴突，计算则可以类比为细胞核，如图 1.11 所示。

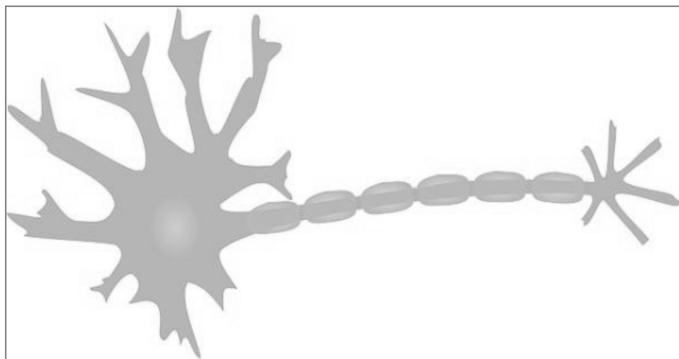


图 1.11　神经元

3．感知机

感知机是一个简单的线形二进制分类器。它接收输入和与其相连的权重（表示输入变量的相对重要性），将它们结合来产生输出。输出接下来被用于分类。感知机已经存在很长一段时间了，最早的使用可追溯到 1950 年代，其中一个也应用在早期的人工神经网络之中，如图 1.12 所示。

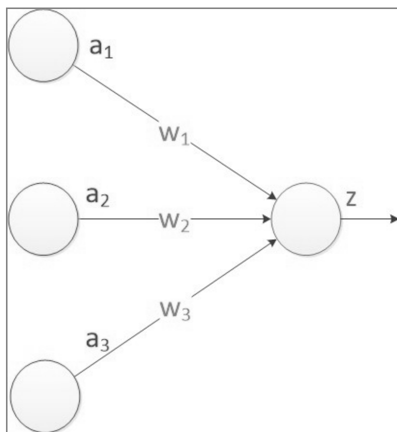


图 1.12　早起的人工神经网络图

4．多层感知机

一个多层感知机（MLP）由几个含有全邻接层的感知机组成，形成一个简单的前馈神经网络。多层感知机在非线性激活函数上有许多好处，这些都是单层感知器不具备的。

5．前馈神经网络

前馈神经网络（Feedforward Neural Network），简称前馈网络，是人工神经网络的一种，如图 1.13 所示。在此种神经网络中，各神经元从输入层开始，接收前一级输入，并输入到下一级，直至输出层。整个网络中无反馈，可用一个有向无环图表示。前馈神经网络是最早被提出的人工神经网络，也是最简单的人工神经网络类型。按照前馈神经网络的层数不同，可以将其划分为单层前馈神经网络和多层前馈神经网络。常见的前馈神经网络有感知机（Perceptrons）、BP 神经网络（Back Propagation）、RBF 径向基网络（Radial Basis Function）等。

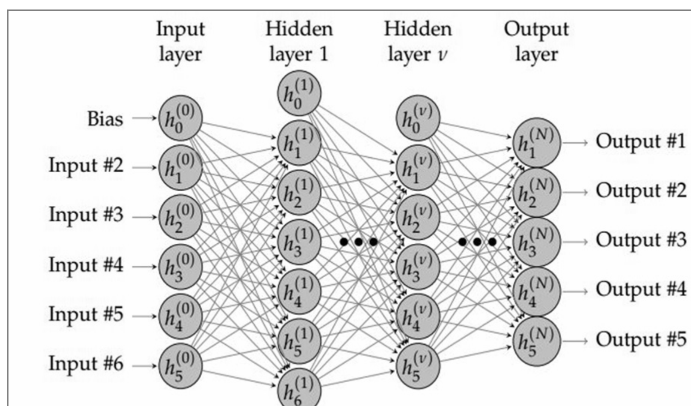


图 1.13　前馈神经网络

6．循环神经网络

和上文所提到的前馈神经网络不同，循环神经网络的连接构成有向循环。这种双向流动允许用内部时间状态表示，继而允许序列处理。并且值得注意的是，它提供了用于识别语音和手写的必要能力。

7．卷积神经网络

卷积神经网络是一种特殊的深层的神经网络模型，卷积神经网络（Convolutional Neural Network，CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于大型图像处理有出色表现，如图 1.14 所示。

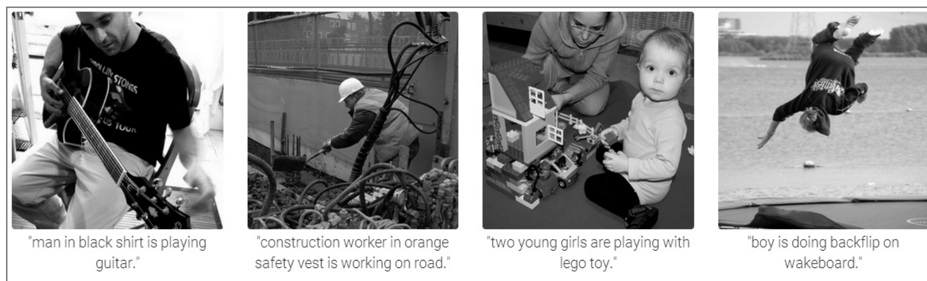


图 1.14　循环神经网络与卷积神经网络生成图片标题

它包括卷积层（Convolutional layer）和池化层（Pooling layer）。它的特殊性体现在两个方面，一方面它的神经元间的连接是非全连接的，另一

方面同一层中某些神经元之间的连接的权重是共享的（即相同的）。它的非全连接和权值共享的网络结构使之更类似于生物神经网络，降低了网络模型的复杂度（对于很难学习的深层结构来说，这是非常重要的），减少了权值的数量。

CNN 主要用来识别位移、缩放及其他形式扭曲不变性的二维图形。由于 CNN 的特征检测层通过训练数据进行学习，所以在使用 CNN 时，避免了显示的特征抽取，而隐式地从训练数据中进行学习；再者由于同一特征映射面上的神经元权值相同，所以网络可以并行学习，这也是卷积网络相对于神经元彼此相连网络的一大优势。

卷积神经网络以其局部权值共享的特殊结构在语音识别和图像处理方面有着独特的优越性，其布局更接近于实际的生物神经网络，权值共享降低了网络的复杂性，特别是多维输入向量的图像可以直接输入网络这一特点避免了特征提取和分类过程中数据重建的复杂度，如图 1.15 所示。

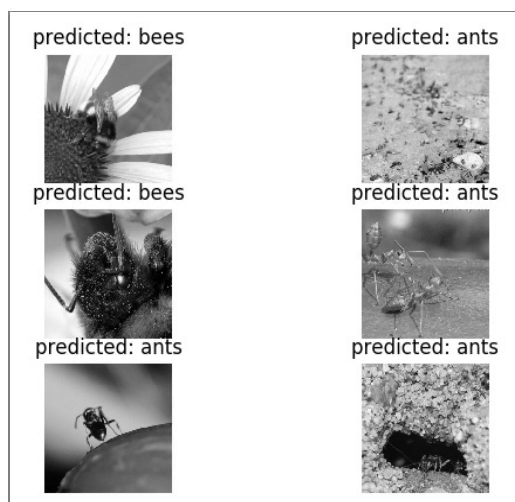


图 1.15　PyTorch 官网提供的数据图片：用神经网络来预测图片中的动物

8．自编码模型

Auto-Encoder（AE）是 20 世纪 80 年代晚期提出的，它是一种无监督学习算法，使用了反向传播算法，让目标值等于输入值。基本的 AE 可视

为一个三层神经网络结构：一个输入层、一个隐藏层和一个输出层，其中输出层与输入层具有相同的规模。自编码器并不是一个真正的无监督学习的算法，而是一个自监督的算法。自监督学习是监督学习的一个实例，其标签产生自输入数据，如图 1.16 所示。

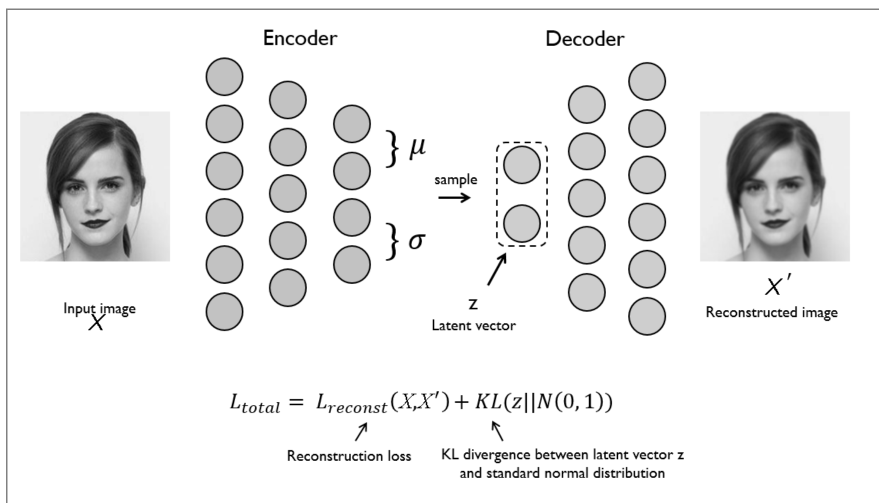


图 1.16　自编码模型

9．对抗生成网络

GAN 启发自博弈论中的二人零和博弈（Two-player Game），GAN 模型中的两位博弈方分别由生成式模型（Generative Model）和判别式模型（Discriminative Model）充当，其应用如图 1.17 所示。生成模型 G 捕捉样本数据的分布，用服从某一分布（均匀分布、高斯分布等）的噪声 Z 生成一个类似真实训练数据的样本，追求效果是越像真实样本越好；判别模型 D 是一个二分类器，估计一个样本来自于训练数据（而非生成数据）的概率，如果样本来自真实的训练数据，D 输出大概率，否则，D 输出小概率。可以做如下类比：生成网络 G 好比假币制造团伙，专门制造假币，判别网络 D 好比警察，专门检测使用的货币是真币还是假币，G 的目标是想方设法生成和真币一样的货币，使得 D 判别不出来，D 的目标是想方设法检测出来 G 生成的是假币。

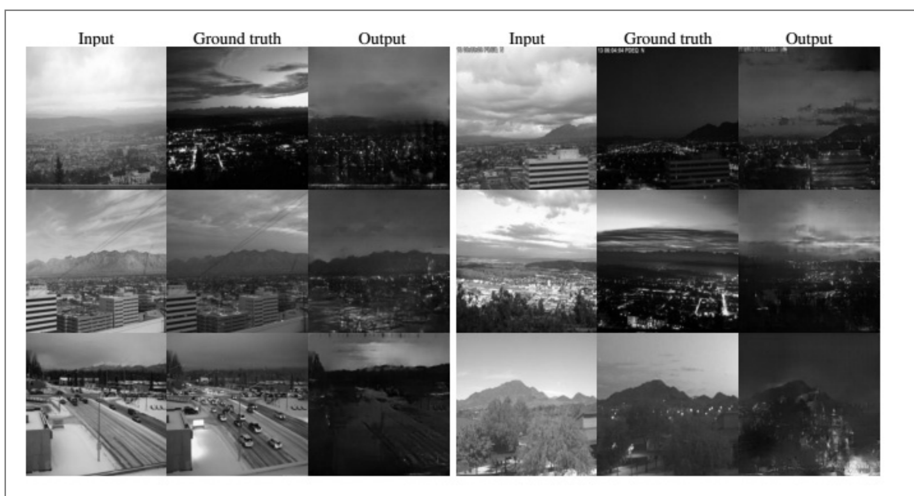


图 1.17　用生成对抗网络生成图片

10．深度学习的数据集、训练集和测试集选择

深度学习是一个典型的迭代过程，需要多次重复循环，为训练数据找到一个称心的神经网络。我们都知道对数据进行多次的迭代，需要很大的计算量，为此我们在训练数据的时候需要创建高质量的训练集和测试集，有助于我们提高效率。通常对于一个完整的数据，我们把一部分数据作为训练集，一部分作为测试集。

在传统的机器学习模型中，我们收集的数据可能是几千条，几万条，或者更多一点，但远远达不到海量数据的要求。通常是将所有的数据分为两个部分，70%作为训练集，30%作为测试集。在大数据的时代，我们有可能拥有上亿条级别的数据，用深度学习训练海量数据，我们该如何选择训练集和测试集呢？

由于数据量呈百万级别的，测试集的比例占数据总量的比例相对较小，可能取到 1%作为测试集，而此时的测试集数据也有上万条，我们可以选择多种神经网络模型进行训练，测试集的目的就是用结果来测试选择神经网络模型的效果，是对所选定的神经网络模型做出的无偏估计。此时测试集足以满足测试选择的神经网络模型所需求的数据量。我们从中选择测试结果比较优良的神经网络模型作为本次数据训练的模型。

过拟合概念：至于过拟合是什么，就是模型训练时候的误差很小，但在测试的时候误差很大，也就是我们的模型复杂到可以拟合到我们的所有训练样本了，但在实际预测新的样本的时候，糟糕得一塌糊涂。

11. 常见的距离公式

（1）闵可夫斯基距离

闵可夫斯基距离（Minkowski distance）是衡量数值点之间距离的一种非常常见的方法。

闵可夫斯基距离定义为：

$$\left(\sum_{i=1}^{n}|x_i - y_i|^p\right)^{1/p}$$

该距离最常用的 p 是 2 和 1，前者是欧几里得距离（Euclidean distance），后者是曼哈顿距离（Manhattan distance）。假设在曼哈顿街区乘坐出租车从 P 点到 Q 点，白色表示高楼大厦，灰色表示街道，如图 1.18 所示。

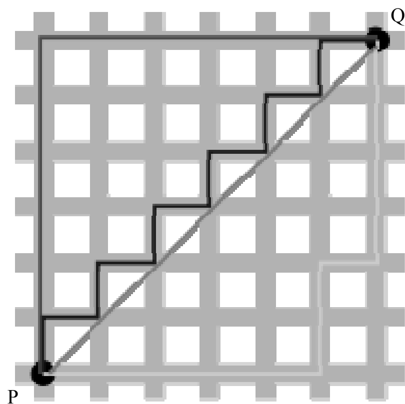


图 1.18　闵可夫斯基距离

斜线表示欧几里得距离，在现实中是不可能的。其他三条折线表示了曼哈顿距离，这三条折线的长度是相等的。

当 p 趋近于无穷大时，闵可夫斯基距离转化成切比雪夫距离（Chebyshev distance）：

$$\lim_{p\to\infty}\left(\sum_{i=1}^{n}\left|x_i-y_i\right|p\right)^{\frac{1}{p}}=\max_{i=1}^{n}\left|x_i-y_i\right|。$$

（2）马氏距离（Mahalanobis distance）是由印度统计学家马哈拉诺比斯（P. C. Mahalanobis）提出的，表示数据的协方差距离。它是一种有效地计算两个未知样本集的相似度的方法。与欧氏距离不同的是它考虑各种特性之间的联系（例如：一条关于身高的信息会带来一条关于体重的信息，因为两者是有关联的。），并且是尺度无关的（Scale-invariant），即独立于测量尺度。对于一个均值为 μ，协方差矩阵为 Σ 的多变量向量，其马氏距离为 $(x-\mu)'\sum^{(-1)}(x-\mu)$ 。

（3）余弦相似度，又称为余弦相似性。通过计算两个向量的夹角余弦值来评估它们的相似度。

假设向量 a、b 的坐标分别为（x_1，y_1）、（x_2，y_2）。则：

$$\cos\theta=\frac{a\bullet b}{\|a\|\|b\|}$$

设向量 $\boldsymbol{A}$=（$\boldsymbol{A}_1$，$\boldsymbol{A}_2$，…，$\boldsymbol{A}_n$），$\boldsymbol{B}$=（$\boldsymbol{B}_1$，$\boldsymbol{B}_2$，…，$\boldsymbol{B}_n$）。

推广到多维：

$$\cos\theta=\frac{x_1x_2+y_1y_2}{\sqrt{x_1^2+y_1^2}\times\sqrt{x_2^2+y_2^2}}$$

$\cos\theta=\frac{\sum_1^n(A_i\times B_i)}{\sqrt{\sum_1^n A_i^2}\times\sqrt{\sum_1^n B_i^2}}$ 余弦值的范围在[–1，1]之间，值越趋近于 1，代表两个向量的方向越趋近于 0，它们的方向更加一致。相应的相似度也越高。

（4）数据标准化处理

数据标准化（归一化）处理，是为了消除指标之间的量纲影响，以解决数据指标之间的可比性。原始数据经过数据标准化处理后，各指标处于同一数量级，适合进行综合对比评价。标准化就是一种对样本数据在不同维度上进行一个伸缩变化，也就是不改变原始数据的信息（分布）。这样的好处就是在进行特征提取时，忽略掉不同特征之间的一个度量，而保留样本在各个维度上的信息（分布）。

以下是两种常用的归一化方法。

a．Min-Max 标准化（Min-Max Normalization）

也称为离差标准化，是对原始数据的线性变换，使结果值映射到[0，1]之间。转换函数如下：

$$X^* = \frac{X - \min}{\max - \min}$$

其中 max 为样本数据的最大值，min 为样本数据的最小值。这种方法有个缺陷就是当有新数据加入时，可能导致 max 和 min 的变化，需要重新定义。

b．Z-score 标准化方法

这种方法给予原始数据的均值（mean）和标准差（standard deviation）进行数据的标准化。经过处理的数据符合标准正态分布，即均值为 0，标准差为 1，转化函数为 $X^* = \frac{X - \mu}{\sigma}$，其中 u 为所有样本数据的均值，$\sigma$ 为所有样本数据的标准差。

（5）正则化

为了防止过拟合现象，我们加入了正则化项，常用的有 L1 范数和 L2 范数。

常用的向量的范数如下。

L0 范数：$||x||0$ 为 x 向量各个非零元素的个数。

L1 范数：$||x||1$ 为 x 向量各个元素绝对值之和，也叫“稀疏规则算子”（Lasso Regularization）。

L2 范数：$||x||2$ 为 x 向量各个元素平方和的 1/2 次方，L2 范数又称 Euclidean 范数或者 Frobenius 范数。在回归里面，有人把有它的回归叫“岭回归”（Ridge Regression），有人也叫它“权值衰减（Weight Decay）”。

Lp 范数：$||x||$为 x 向量各个元素绝对值 p 次方和的 $1/p$ 次方。

L∞范数：$||x||$为 x 向量各个元素绝对值最大那个元素的绝对值。

常用的正则化项除了 L1 范数和 L2 范数外，还有一种名为 Dropout 的

方法。在 2012 年文献 *Improving neural networks by preventing co-adaptation of feature detectors* 中指出，在每次训练神经网络的时候，让一半的特征检测器（也叫神经元）停止工作，这样可以提高网络的泛化能力，Hinton 又把它称之为 Dropout。Hinton 认为过拟合，可以通过阻止某些特征的协同作用来缓解。在每次训练的时候，每个神经元有百分之五十的概率被移除，这样可以让一个神经元的出现依赖于另外一个神经元。

Dropout 以概率 p 舍弃神经元并让其他神经元以概率 q=1–p 保留。每个神经元被关闭的概率是相同的。图 1.19 为 Dropout 的可视化表示，左边是应用 Dropout 之前的网络，右边是应用了 Dropout 的同一个网络。

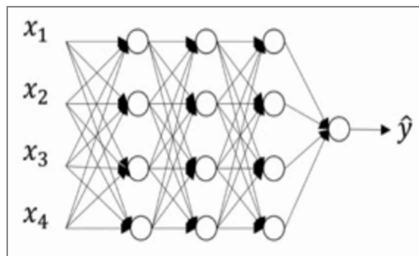


图 1.19　没有经过舍弃神经元的模型

对于图 1.20 有可能出现过拟合且含有诸多参数的隐藏层，我们可以调整 Dropout 的概率为较小的值来防止过拟合现象。在计算机视觉领域中，为防止过度拟合，最常用的方法就是 Dropout。

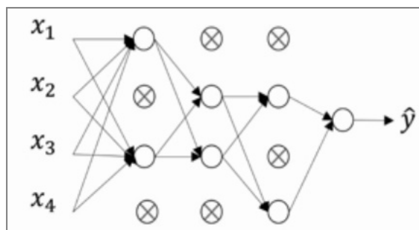


图 1.20　经过舍弃神经元的模型

（6）成本函数

在训练神经网络时，必须评估网络输出的正确性。众所周知，预期上正确的训练输出数据和实际的训练输出是可比拟的。成本函数便能测量实际和训练输出之间的差异。实际和预期输出之间的零成本将意味着训练神

经网络成为可能。

BGD 即 Batch Gradient Descent。在训练中，每一步迭代都使用训练集的所有内容。也就是利用现有参数对训练集中的每一个输入生成一个估计输出，然后跟实际输出比较，统计所有误差，求平均以后得到平均误差，以此来作为更新参数的依据。

由于每一步都利用了训练集中的所有数据，因此当损失函数达到最小值以后，能够保证此时计算出的梯度为 0，换句话说，就是能够收敛。因此，使用 BGD 时不需要逐渐减小学习速率。由于每一步都要使用所有数据，因此随着数据集的增大，运行速度会越来越慢。在批量梯度下降法中，因为每次都遍历了完整的训练集，其能保证结果为全局最优，但是也因为我们需要对每个参数求偏导，且在对每个参数求偏导的过程中还需要对训练集遍历一次，当训练集很大时，这个计算量是惊人的！

所以，为了提高速度，减少计算量，提出了 SGD 随机梯度下降的方法，该方法每次随机选取一个样本进行梯度计算，大大降低了计算成本。

SGD 全名 Stochastic Gradient Descent，即随机梯度下降。即随机抽取一批样本，以此为根据来更新参数。随机梯度下降算法和批量梯度下降的不同点在于其梯度是根据随机选取的训练集样本来决定的，其每次对 theta 的更新，都是针对单个样本数据，并没有遍历完整的参数。当样本数据很大时，可能到迭代完成，也只不过遍历了样本中的一小部分。因此，其速度较快，但是其每次的优化方向不一定是全局最优的，但最终的结果是在全局最优解的附近。

12．Momentum

Momentum 借用了物理中的动量概念，即前几次的梯度也会参与运算。为了表示动量，引入了一个新的变量 v（velocity）。v 是之前的梯度的累加，但是每回合都有一定的衰减。前后梯度方向一致时，能够加速学习。前后梯度方向不一致时，能够抑制震荡。

Nesterov Momentum 这是对之前的 Momentum 的一种改进，大概思路就是，先对参数进行估计，然后使用估计后的参数来计算误差。

13．AdaGrad

AdaGrad 可以自动变更学习速率，只需要设定一个全局的学习速率，但是这并非是实际学习速率，实际的速率是与以往参数的模之和的开方成反比的。

它能够实现学习率的自动更改。如果这次梯度大，那么学习速率衰减就快一些；如果这次梯度小，那么学习速率衰减就慢一些。但仍然要设置一个变量。

经验表明，在普通算法中也许效果不错，但在深度学习中，深度过深时会造成训练提前结束。

14．RMSProp

RMSProp 通过引入一个衰减系数 r，让 r 每回合都衰减一定比例，类似于 Momentum 中的做法。相比于 AdaGrad，这种方法很好地解决了深度学习中过早结束的问题。

适合处理非平稳目标，对于卷积神经网络效果很好，又引入了新的超参，衰减系数 ρ，其实 RMSprop 依然依赖于全局学习率。

RMSprop 算是 Adagrad 的一种发展，Adadelta 的变体，效果趋于二者之间。

15．Adam

Adam（Adaptive Moment Estimation）本质上是带有动量项的 RMSprop，它利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习速率。Adam 的优点主要在于经过偏置校正后，每一次迭代学习速率都有个确定范围，使得参数比较平稳。结合了 Adagrad 善于处理稀疏梯度和 RMSprop 善于处理非平稳目标的优点，对内存需求较小，为不同的参数计算不同的自适应学习速率，也适用于大多非凸优化，适用于大数据集和高维空间。

如何快速入门深度学习呢？

刚开始接触机器学习都认为机器学习很难，并不是因为数学难，而是因为不知道选择什么模型适合什么类型的数据。快速有效地选择模型来训练数据是现代机器学习中的必备技能，但机器学习的模型选择相比普通程

序要难很多：候选错误空间大、调试周期长。我们都知道推动机器学习研究进步的科学本身很困难，需要创新、实验和坚持。把已知的机器学习模型应用到实际工作中也是一件困难的事情。我们选择了什么样的机器学习模型需要对每个算法的优势和劣势都了如指掌，当然这类知识构建的困难是计算机所有领域都存在的，不仅仅是机器学习。

对于深度学习而言，本身并不难，难的是你吃透问题，如何用深度学习的逻辑去思考你自己的问题，有针对性地设计模型；难的是你有分析问题和结果的能力，遇到负面结果不是抓瞎。能解决问题就是最好的模型。

如何才能快速入门机器学习呢？

对于入门而言，并不需要很多的数学知识。也许很多人在翻看任何一本机器学习的书时，看到一推数学公式就开始打退堂鼓了。大学期间学习的“线性代数”“高数”“概率论与数理统计”就可以让你入门了。其中重点部分为线性代数：在机器学习中，线性代数随处可见。主成分分析（PCA）、奇异值分解（SVD）、矩阵的特征分解、LU 分解、QR 分解、对称矩阵、正交化和正交归一化、矩阵的运算、分解、向量空间和范数等，这些都是理解机器学习中所使用的优化方法。

2

第 2 章 PyTorch 环境安装

PyTorch 目前支持 Linux 和 Mac 平台，在以后的章节中我们使用 Ubuntu 平台进行 PyTorch 的学习。Ubuntu 基于 Debian 发行版和 GNOME 桌面环境，它的界面友好，使用命令（apt-get）也非常简单，很多情况下只需一行命令就可以安装第三方的软件包。

在 PyTorch 官网上每种平台提供了 Conda、pip、Source 三种安装方式，同时也可以根据有无 GPU 进行 CUDA 安装，在这里以 Ubuntu16.04 操作系统为例，使用 pip 命令的方式进行安装。

2.1 基于Ubuntu环境的安装

Ubuntu 是本书内容所依赖的环境，本书使用的是 Ubuntu16.04 LTS 桌面版，机器配置如图 2.1 所示。

本书使用 Ubuntu 操作系统的下载地址为 https://www.ubuntu.com/download/alternative-downloads，安装如图 2.2 所示。

Ubuntu 是一个以桌面应用为主的开源 GNU/Linux 操作系统，安装成功后已经默认安装 Python 环境，默认安装的 Python 版本为 Python 2.7。



图 2.1　Ubuntu16.04 LTS 桌面版

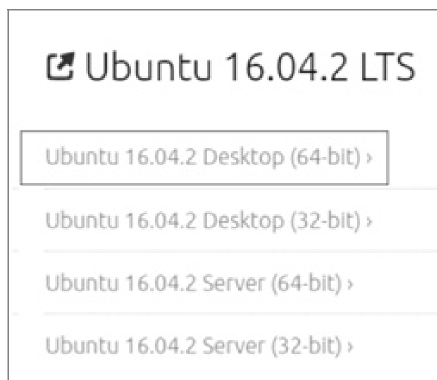


图 2.2　选择安装 Ubuntu16.04 LTS 64-bit

Python 2 与 Python 3 虽然语法结构有些类似，但是却不能完全兼容。这里主要介绍的语言开发环境是 Python 3，原因如下。

（1）目前，Python 2 的绝大部分开源框架都提供了对 Python 3 的支持，并且一些新的开源框架如 TensorFlow 等只提供了对 Python 3 的支持。

（2）对于 Python 2，官方只支持到 2020 年，而 Python 的新开源框架往往不会对即将被淘汰的语言提供太多的支持，而且从 Python 2 到 Python 3 是做了大量性能上的改进，符合语言发展规律，Python 3 会是以后的主流。

2.1.1　安装 Anaconda

PyTorch 安装起来很方便，根据系统环境选择 pip 安装或者 Conda 安装（需要操作系统先安装 Anaconda 环境），如果使用 GPU 需要先安装 CUDA，根据版本选择即可，免去了类似安装 Caffe 依赖库、编译 Caffe 的很多麻烦，如图 2.3 所示。

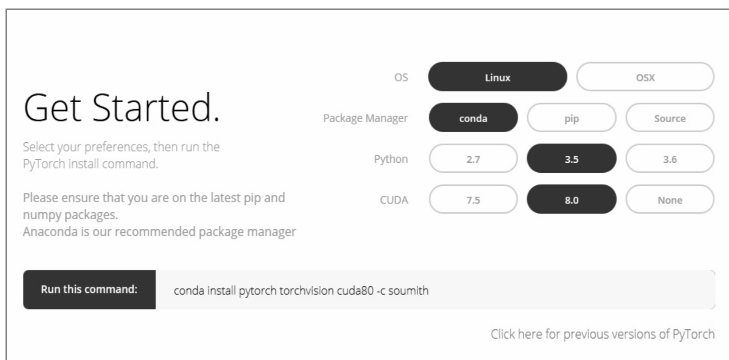


图 2.3　Anaconda 适应安装环境

本书选用 Conda 方式安装 PyTorch，首先安装 Anaconda，Anaconda 是一个 PyTorch 包管理工具，能够在同一机器上创建多个互不影响的 Python 环境。

去 Anaconda 官网下载安装包，下载地址：http://continuum.io/downloads。页面如图 2.4 所示。

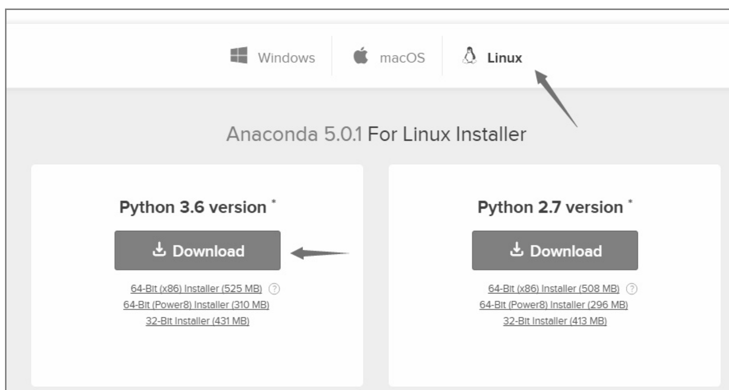


图 2.4　下载 Anaconda

安装时，会发现有两个不同版本的 Anaconda，分别对应 Python 2.7 和 Python 3.6，两个版本其实除了这点区别外其他都一样。根据自己的系统选择相应版本进行下载，下载之后点击运行就可以安装了，和一般软件安装毫无二致，无须编译。

本书使用的是 Python 3.6 环境，所以选择 Python 3.6 版本对应的 Anconda 进行下载。使用 root 账号进行安装。本书下载的 Anaconda 文件名为 Anaconda3-5.0.1-Linux- x86_64.sh。使用 bash 命令进行安装。

```
$ bash Anaconda3-5.0.1-Linux-x86_64.sh
```

安装成功后，Anaconda 3 默认安装在/root/anacondas 目录下，还需要配置 /etc/profile。编辑/etc/profile 文件，把下面的 Anconda 3 的安装路径添加进去。

```
export PATH=$PATH:.:/root/anaconda3/bin:
```

配置后输入命令，使配置文件生效。

```
$ source /etc/profile
```

可以输入常用的 Conda 命令查看是否生效。

（1）查询安装信息

```
$ conda info
```

（2）查询当前已经安装的库

```
$ conda list
```

（3）安装库（*代表库 lib 名称）

```
$ conda install ***
```

2.1.2 设置国内镜像

官方下载更新工具包的速度很慢，所以继续添加清华大学 TUNA 提供的 Anaconda 仓库镜像，在终端中输入如下命令进行添加。

```
$ conda config --add channels
```

```
https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/ $ conda
config --set show_channel_urls yes
```

2.2　Conda命令安装PyTorch

首先要进行 PyTorch 适应安装环境，如图 2.5 所示。

Get Started.

Select your preferences, then run the PyTorch install command.

Please ensure that you are on the latest pip and numpy packages.
Anaconda is our recommended package manager

OS	Linux	OSX	
Package Manager	conda	pip	Source
Python	2.7	3.5	3.6
CUDA	7.5	8.0	None

Run this command: conda install pytorch torchvision cuda80 -c soumith

图 2.5　PyTorch 适应安装环境之 Conda

然后再使用 Conda 命令进行装。

```
$ conda install pytorch torchvision cuda80 -c soumith
```

2.3　pip命令安装PyTorch

也可以使用 pip 命令进行安装，安装的 Anconda 3 环境中，集成了 Python 3.5 的环境，如图 2.6 所示。

Get Started.

Select your preferences, then run the PyTorch install command.

Please ensure that you are on the latest pip and numpy packages.
Anaconda is our recommended package manager

OS	Linux	OSX	
Package Manager	conda	pip	Source
Python	2.7	3.5	3.6
CUDA	7.5	8.0	None

Run this command: pip3 install http://download.pytorch.org/whl/cu80/torch-0.2.0.post3-cp35-cp35m-manylinux1_x86_64.whl
pip3 install torchvision

图 2.6　PyTorch 适应安装环境之 pip

首先，下载 torch-0.2.0.post3-cp35-cp35m-manylinux1_x86_64.whl

```
    http://download.pytorch.org/whl/cu80/torch-0.2.0.post3-cp35-c
p35m-manylinux1_x86_64.whl
```

然后，再使用 pip 命令安装 whl 和必要的模块

```
    $ pip3 install
/cu80/torch-0.2.0.post3-cp35-cp35m-manylinux1_x86_64.whl
    $ pip3 install torchvision  -i
https://pypi.tuna.tsinghua.edu.cn/simple torchvision
```

pip 3 命令中的参数–i 表示使用的是清华大学 TUNA 提供的镜像，因为 PyTorch 的安装包部署在国外的服务器上。

2.4　配置CUDA

为了使用 GPU 进行计算的加速，还需要安装 CUDA，直接从官网上下载对应版本的 CUDA，CUDA 的下载地址如下。

https://developer.nvidia.com/cuda-downloads

我们下载 Ubuntu 版本下的 CUDA 安装包，这是一个 Linux 下 64 位的 deb 格式的安装包。我们使用最新版的 Linux 64 位安装包，cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64.deb。

然后使用 dpkg 命令安装 CUDA 的 deb 格式的安装包，如图 2.7 所示。

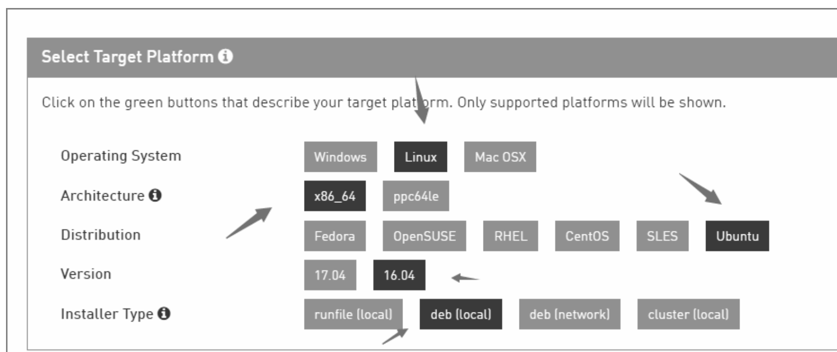


图 2.7　安装 CUDA

```
$ dpkg -i cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64.deb
```

安装成功后，还有附加的库需要安装。

```
$ apt-get update
$ apt-get install cuda
```

上面 apt-get 安装的 CUDA 会根据版本的不同，而安装相应的库。比如你如果是采用 CUDA 8.0 的 deb，那么此时上面会出现一堆 cuda8.0-之类的文件名的库。

最后再试试是否安装好了 CUDA。

```
$ nvidia-smi
```

如果出现显卡配置信息的话，就安装成功了，如图 2.8 所示。

```
root@pc-desktop:/home/pc# nvidia-smi
Mon Dec  4 16:03:54 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.90                 Driver Version: 384.90                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1050    Off  | 00000000:01:00.0  On |                  N/A |
| 40%   25C    P8    35W /  75W |    178MiB /  1997MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0       900      G   /usr/lib/xorg/Xorg                            136MiB |
|    0      1492      G   compiz                                         39MiB |
+-----------------------------------------------------------------------------+
```

图 2.8　显卡信息

3

第 3 章

PyTorch 基础知识

3.1 张量

在数学里，张量是一种几何实体，或者说广义上的“数量”。张量概念包括标量、向量和线性算子。张量可以用坐标系统来表达，记作标量的数组。

Tensors 类似于 NumPy 的 ndarray，另外还可以在 GPU 上使用 Tensor 来加速计算。图 3.1 是 Tensor 常见的类型。

Torch defines seven CPU tensor types and eight GPU tensor types:

Data type	CPU tensor	GPU tensor
32-bit floating point	torch.FloatTensor	torch.cuda.FloatTensor
64-bit floating point	torch.DoubleTensor	torch.cuda.DoubleTensor
16-bit floating point	torch.HalfTensor	torch.cuda.HalfTensor
8-bit integer (unsigned)	torch.ByteTensor	torch.cuda.ByteTensor
8-bit integer (signed)	torch.CharTensor	torch.cuda.CharTensor
16-bit integer (signed)	torch.ShortTensor	torch.cuda.ShortTensor
32-bit integer (signed)	torch.IntTensor	torch.cuda.IntTensor
64-bit integer (signed)	torch.LongTensor	torch.cuda.LongTensor

图 3.1　Tensor 常见的类型

构造一个 4×5 的矩阵，本例文件名为 PyTorch/Chapter03/pt01_tensor.py。

```
import torch
z = torch.Tensor(4, 5)
print(z)
```

结果显示如下：

```
 0.0000  0.0000  0.0000  0.0000  0.0000
 0.1839  0.0000  0.0000  0.0000  0.0000
 0.0000  0.8428  0.9081  0.5882  0.0000
 0.0000  0.0000  0.0000  0.0000  0.0000
[torch.FloatTensor of size 4x5]
```

两个矩阵进行加法操作。

```
y = torch.rand(4, 5)#产生一个 4 行 5 列的矩阵
print(z + y)
```

结果显示如下：

```
3.1953e-01  1.1411e-01  4.8911e+08  3.6180e-01  7.0273e-01
 9.8657e-01  4.4887e-01  7.4852e-02  5.9674e-01  5.1458e-03
 5.2158e-01  1.5118e-01  7.3711e-01  4.8894e-01  5.8435e-01
 7.9632e-01  4.8496e-01  4.7753e-01  9.4765e-02  2.2932e-01
[torch.FloatTensor of size 4x5]
```

两个矩阵加法的另一种表达形式。

```
print(torch.add(z, y))
```

结果显示如下：

```
3.1953e-01  1.1411e-01  4.8911e+08  3.6180e-01  7.0273e-01
 9.8657e-01  4.4887e-01  7.4852e-02  5.9674e-01  5.1458e-03
 5.2158e-01  1.5118e-01  7.3711e-01  4.8894e-01  5.8435e-01
 7.9632e-01  4.8496e-01  4.7753e-01  9.4765e-02  2.2932e-01
[torch.FloatTensor of size 4x5]
```

将 Tensor 转换为 numpy 数组。

```
b = z.numpy()
print(b)
```

结果显示如下：

```
[[  8.10158129e-38   4.56949416e-41   8.10158129e-38
4.56949416e-41
       1.40129846e-45]
 [  9.80908925e-45   2.48065457e-11   3.08201584e-41
0.00000000e+00
       0.00000000e+00]
 [  0.00000000e+00   1.40129846e-45   8.96831017e-44
0.00000000e+00
       4.62428493e-44]
 [  0.00000000e+00   4.72502037e-11   3.08201584e-41
4.70757322e-11
       3.08201584e-41]]
```

将 NumPy 数组转换为 Torch 张量。本例文件名为 PyTorch/Chapter03/pt02_tensor.py。

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

结果显示如下：

```
[ 2.  2.  2.  2.  2.]

 2
 2
 2
 2
 2
[torch.DoubleTensor of size 5]
```

torch.squeeze（input，dim=None，out=None），将输入张量形状中的 1 去除并返回。当给定 dim 时，那么挤压操作只在给定维度上。

```
>>> x = torch.zeros(2,1,2,1,2)
>>> x.size()
(2L, 1L, 2L, 1L, 2L)
>>> y = torch.squeeze(x)
>>> y.size()
(2L, 2L, 2L)
>>> y = torch.squeeze(x, 0)
>>> y.size()
(2L, 1L, 2L, 1L, 2L)
>>> y = torch.squeeze(x, 1)
>>> y.size()
(2L, 2L, 1L, 2L)
```

3.2　数学操作

PyTorch 内置大量的数学操作函数，如加法操作函数，绝对值操作函数，随机生成数组函数等。

本例文件名为 PyTorch/Chapter03/pt03_math.py。

torch.abs（input，out=None），计算输入张量的每个元素的绝对值。

```
>>> torch.abs(torch.FloatTensor([-1, -2, 3]))
```

torch.acos（input，out=None），返回一个新张量，包含输入张量每个元素的反余弦。

torch.add（input，value，out=None），对输入张量 input 逐元素加上标量值 value，并返回结果得到一个新的张量 out。

```
>>> a = torch.randn(4)
>>> a
 0.4050
-1.2227
 1.8688
-0.4185
[torch.FloatTensor of size 4]
>>> torch.add(a, 20)
```

```
 20.4050
 18.7773
 21.8688
 19.5815
[torch.FloatTensor of size 4]
```

3.3 数理统计

PyTorch 提供了大量数理统计函数，方便读者使用。

本例文件名为 PyTorch/Chapter03/pt04_math.py。

torch.mean（input），返回输入张量所有元素的均值。

```
>>> a = torch.randn(1, 3)
>>> a

-0.2946 -0.9143  2.1809
[torch.FloatTensor of size 1x3]

>>> torch.mean(a)
0.32398951053619385
```

torch.mean（input，dim，out=None），返回输入张量给定维度 dim 上每行的均值。

```
>>> a = torch.randn(4, 4)
>>> a

-1.2738 -0.3058  0.1230 -1.9615
 0.8771 -0.5430 -0.9233  0.9879
 1.4107  0.0317 -0.6823  0.2255
-1.3854  0.4953 -0.2160  0.2435
[torch.FloatTensor of size 4x4]

>>> torch.mean(a, 1)
```

```
-0.8545
 0.0997
 0.2464
-0.2157
[torch.FloatTensor of size 4x1]
```

3.4　比较操作

PyTorch 提供了比较操作函数。

本例文件名为 PyTorch/Chapter03/pt05_operator.py。

torch.eq（input，other，out=None），比较元素相等性。第二个参数可为一个数或与第一个参数同类型形状的张量。

```
>>> torch.eq(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
1  0
0  1
[torch.ByteTensor of size 2x2]
torch.equal(tensor1, tensor2)
```

如果两个张量有相同的形状和元素值，则返回 True，否则 False。

```
>>> torch.equal(torch.Tensor([1, 2]), torch.Tensor([1, 2]))
True
```

torch.ge（input，other，out=None），逐元素比较 input 和 other。如果两个张量有相同的形状和元素值，则返回 True，否则 False。第二个参数可以为一个数或与第一个参数相同形状和类型的张量。

```
>>> torch.ge(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 1  1
 0  1
[torch.ByteTensor of size 2x2]
```

torch.gt（input，other，out=None），逐元素比较 input 和 other。如果两个张量有相同的形状和元素值，则返回 True，否则 False。第二个参数可以为一个数或与第一个参数相同形状和类型的张量。

```
>>> torch.gt(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 0  1
 0  0
[torch.ByteTensor of size 2x2]
```

第 4 章

4

简单案例入门

4.1 线性回归

线性回归是利用数理统计中的回归分析，来确定两种或两种以上变量间相互依赖的定量关系的一种统计分析方法，运用十分广泛。分析按照自变量和因变量之间的关系类型，可分为线性回归分析和非线性回归分析。

回归分析中，只包括一个自变量和一个因变量，且二者的关系可用一条直线近似表示，这种回归分析称为一元线性回归分析。如果回归分析中包括两个或两个以上的自变量，且因变量和自变量之间是线性关系的，则称为多元线性回归分析。

线性模型基本形式

线性回归属于回归算法，表达监督学习的过程。通过属性的线性组合来预测函数：

$$f(x) = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b$$

一般向量形式写成：

$$f(x) = \boldsymbol{w}^{\mathrm{T}} x + b$$

其中的 $\boldsymbol{w}$=（w_1；w_2；…；w_d）。

x_1，x_2，…，x_k 是一组独立的预测变量。

w_1，w_2，…，w_k 为模型从训练数据中学习到的参数，或赋予每个变量的“权值”。

b 也是一个学习到的参数，这个线性函数中的常量也称为模型的偏置（Bias）。

线性回归的目标是找到一个与这些数据最为吻合的线性函数，用来预测或者分类，主要解决线性问题。在有监督学习问题中，线性回归是一种最简单的建模手段。给定一个数据点集合作为训练集，线性回归的目标是找到一个与这些数据最为吻合的线性函数。对于 2D 数据，这样的函数对应一条直线。

一般来说，线性回归都可以通过最小二乘法求出其方程。线性回归属于监督学习，因此方法和监督学习应该是一样的，先给定一个训练集，根据这个训练集学习出一个线性函数，然后测试这个函数训练得好不好（即此函数是否足够拟合训练集数据），挑选出最好的函数（Cost Function 最小）即可。Cost Function 越小的函数，说明对训练数据拟合得越好。

接下来我们来重点讲讲如何用 PyTorch 写一个简单的线性回归函数。

本例文件名为 PyTorch/Chapter04/pt01_linear regression.py。

首先我们导入 Torch 模块库，Matplotlib 画图模块库。

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from torch.autograd import Variable
input_size = 1
output_size = 1
learning_rate = 0.001
```

我们再生成数据。

```
    xtrain = np.array([[2.3], [4.4], [3.7], [6.1], [7.3], [2.1],[5.6],
[7.7], [8.7], [4.1], [6.7], [6.1], [7.5], [2.1], [7.2], [5.6], [5.7],
[7.7], [3.1]], dtype=np.float32)
    #xtrain 生成矩阵数据
    ytrain = np.array([[3.7], [4.76], [4.], [7.1], [8.6], [3.5],[5.4],
[7.6], [7.9], [5.3], [7.3], [7.5], [8.5], [3.2], [8.7], [6.4], [6.6],
[7.9], [5.3]], dtype=np.float32)
    #ytrain 生成矩阵数据
    plt.figure()
    #画散点图
    plt.scatter(xtrain,ytrain)
    plt.xlabel('xtrain')
    #x 轴名称
    plt.ylabel('ytrain')
    #y 轴名称
    #显示图片
    plt.show()
```

运行脚本，输出结果如图 4.1 所示。

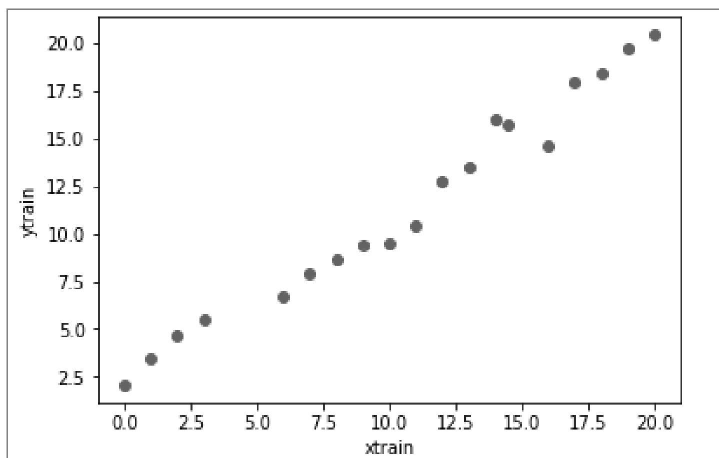


图 4.1　线性回归输出结果

构建线性回归模型如下。

```
    class LinearRegression(nn.Module):
```

```
    def __init__(self, input_size, output_size):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        out = self.linear(x)
        return out
```

这里的 nn.Linear 表示的是 $y=w \times x+b$，里面的两个参数都是 1，表示的是 x 是 1 维，y 也是 1 维。

```
criterion = nn.MSELoss()
#定义 criterion 误差函数，需要将 model 的参数 model.parameters()传进去。对于这种简单的模型，将采用总平方误差，即模型对每个训练样本的预测值与期望输出之差的平方的总和。
```

optimizer = torch.optim.SGD（model.parameters()，lr=learning_rate）

这里使用的是最小二乘 Loss，但是我们做分类问题更多使用的是 Cross Entropy Loss，交叉熵。优化函数使用的是随机梯度下降。优化函数代表我们要通过什么方式去优化我们需要学习的值，这个例子里指的是 w 和 b，优化函数的种类有很多，大家到官网查阅，平时我们使用比较多的是 Gradient Descent Optimizer 和 Adam Optimizer 等，这里我们选用最常用，也是最基本的 Gradient Descent Optimizer（梯度下降），后面传入的值是学习效率。一般是一个小于 1 的数。越小收敛越慢，但并不是越大收敛越快，取值太大甚至可能不收敛了。

我们接着开始训练 1000 次。

```
num_epochs=1000#训练 1000 次
for epoch in range(num_epochs):
    inputs = Variable(torch.from_numpy(xtrain))
    targets = Variable(torch.from_numpy(ytrain))
#Torch 能将产生的 Tensor 放在 GPU 中加速运算，就像 NumPy 会把 array 放在 CPU 中加速运算。Torch 能和 NumPy 能很好地兼容。使用 torch.from_numPy()和 NumPy 函数，将 NumPy 转换成 Tensor。这样就能自由地转换 NumPy array 和 Torch Tensor 了。
    optimizer.zero_grad()
    outputs = model(inputs)
```

```
    # 前向传播
      loss = criterion(outputs, targets)
    # 计算 loss
      loss.backward()

      optimizer.step()
      # 更新参数

      if (epoch+1) % 50 == 0:
         print ('Epoch [%d/%d], Loss: %.4f'
               %(epoch+1, num_epochs, loss.data[0]))    #每隔 50 次
打印一次结果
```

在每次反向传播的时候需要将参数的梯度归零。

训练完成之后我们就可以开始测试模型了。

```
Epoch[50/1000], loss: 3.385932
Epoch[100/1000], loss: 2.111750
Epoch[150/1000], loss: 1.476992
Epoch[200/1000], loss: 1.160495
Epoch[250/1000], loss: 1.002404
Epoch[300/1000], loss: 0.923156
Epoch[350/1000], loss: 0.883154
Epoch[400/1000], loss: 0.862686
Epoch[450/1000], loss: 0.851943
Epoch[500/1000], loss: 0.846042
Epoch[550/1000], loss: 0.842550
Epoch[600/1000], loss: 0.840261
Epoch[650/1000], loss: 0.838570
Epoch[700/1000], loss: 0.837179
Epoch[750/1000], loss: 0.835938
Epoch[800/1000], loss: 0.834774
Epoch[850/1000], loss: 0.833648
Epoch[900/1000], loss: 0.832544
Epoch[950/1000], loss: 0.831451
Epoch[1000/1000], loss: 0.830365
```

经过 1000 次的优化训练，误差缩小至 0.830365。我们画图看一下预测的结果，如图 4.2 所示。

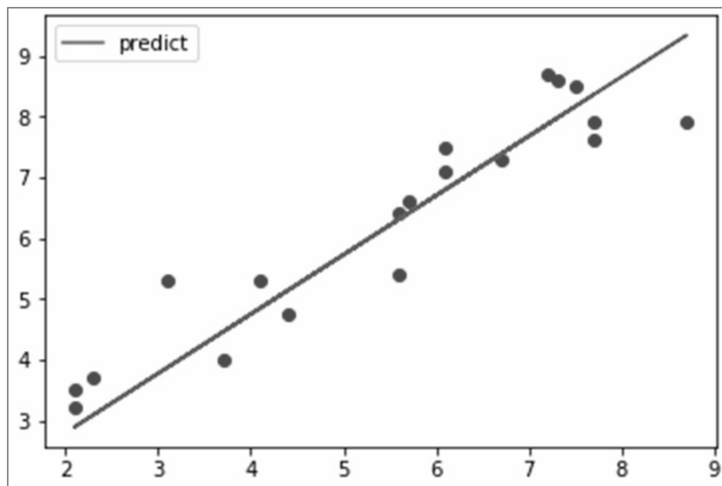


图 4.2　输出预测结果

```
    model.eval()
    predicted = model
(Variable(torch.from_numpy(x_train))).data.numpy()
    plt.plot(x_train, y_train, 'ro')
    plt.plot(x_train, predicted, label='predict')
    plt.legend()
    plt.show()
```

可以看出，预测的结果接近真实值，说明线性模型还是相当不错的。

4.2　逻辑回归

Logistic Regression（逻辑回归）是当前业界比较常用的机器学习方法，用于估计某种事物的可能性。比如某用户购买某商品的可能性，某病人患有某种疾病的可能性，以及某广告被用户点击的可能性等。

1．二分类问题

二分类问题是指预测的 y 值只有两个取值（0 或 1），二分类问题可以

扩展到多分类问题。例如：我们要做一个垃圾邮件过滤系统，X(i)是邮件的特征，预测的 y 值就是邮件的类别，是垃圾邮件还是正常邮件。对于类别我们通常称为正类（Positive Class）和负类（Negative Class），垃圾邮件的例子中，正类就是正常邮件，负类就是垃圾邮件。

2．Logistic 函数

LR 分类器（Logistic Regression Classifier），在分类情形下，经过学习之后的 LR 分类器其实就是一组权值 w_0，w_1，w_2，…，w_m。当测试样本集中的测试数据来到时，这一组权值按照与测试数据线性加和的方式，求出一个 z 值。

$z = w_0 + w_1 \times x_1 + w_2 \times x_2 + \cdots + w_m \times x_m$。（其中 x_1，x_2，...，x_m 是样本数据的各个特征，维度为 m），如果我们忽略二分类问题中 y 的取值是一个离散的取值（0 或 1），我们继续使用线性回归来预测 y 的取值。这样做会导致 y 的取值并不为 0 或 1。逻辑回归使用一个函数来归一化 y 值，使 y 的取值在区间（0，1）内，这个函数称为 Logistic 函数（Logistic Function），也称为 Sigmoid 函数（Sigmoid Function）。

之后按照 Sigmoid 函数的形式求出 g(z)。

$$g(z) = \frac{1}{1 + e^{-z}}$$

由于 Sigmoid 函数的定义域是（-inf，inf），而值域为（0，1）。因此最基本的 LR 分类器适合对两类目标进行分类。

Logistic 函数当 z 趋近于无穷大时，$g(z)$趋近于 1；当 z 趋近于无穷小时，$g(z)$趋近于 0。Logistic 函数的图形如图 4.3 所示。

我们用代码实现逻辑回归。本例文件名为 PyTorch/Chapter04/pt02_logistic regression.py。

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
```

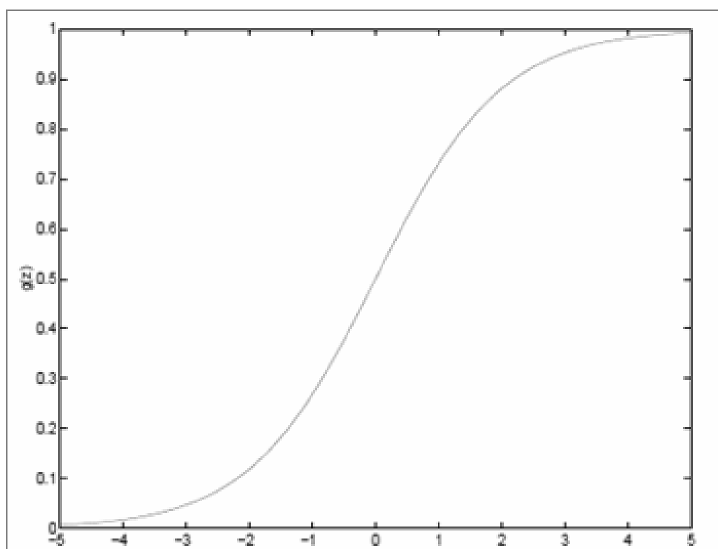


图 4.3　Logistic 函数图像

```
import torchvision.transforms as transforms
from torch.autograd import Variable
#导入 Torch 模块，加载 Torchvision，是专门做图形处理的一个库，加载 dsets
数据集，我们需要使用的是 torchvision.transforms 和 torchvision.datasets
以及 torch.utils.data.DataLoader。

input_size = 784
#图片大小为 784
num_classes = 10
num_epochs = 10
batch_size = 50
#加载批训练的数据个数
learning_rate = 0.001#学习率为 0.001
#设置各种参数
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)
```

参数说明如下。

- root：数据集，存在于根目录 processed/training.pt 和 processed/test.pt 中。

- train：True＝ 训练集，False＝ 测试集。
- download：如果为 True，请从 Internet 下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。
- transform：接收 PIL 映像并返回转换版本的函数/变换。
- target_transform：一个接收目标并转换它的函数/变换。

torchvision.datasets 里面有很多数据类型，里面有官网处理好的数据，比如我们要使用的 MNIST 数据集，可以通过 torchvision.datasets.MNIST() 来得到，还有一个常使用的是 torchvision.datasets.ImageFolder()，这个可以让我们按文件夹来取图片。torchvision.transforms 里面的操作是对导入的图片做处理，比如可以随机取（50，50）这样的窗框大小，或者随机翻转，或者去中间的（50，50）的窗框大小部分等，但是里面必须要用的是 transforms.ToTensor()，这可以将 PIL 的图片类型转换成 Tensor。

```
    test_dataset = dsets.MNIST(root='./data',
                               train=False,
                               transform=transforms.ToTensor())
```

参数说明如下。

- root：数据集，存在于根目录 processed/training.pt 和 processed/test.pt 中。
- train：True＝ 训练集，False＝ 测试集
- download：如果为 True，请从 Internet 下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。
- transform：接收 PIL 映像并返回转换版本的函数/变换。
- target_transform：一个接收目标并转换它的函数/变换。

```
    #测试集
    train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
                                          batch_size=batch_size,
                                          shuffle=True)
```

参数说明如下。

- dataset：加载数据的数据集。
- batch_size：加载批训练的数据个数。

- shuffle：如果为 True，在每个 Epoch 重新排列数据。
- sampler：从数据集中提取样本。
- batch_sampler：一次返回一批索引。
- num_workers：用于数据加载的子进程数。0 表示数据将在主进程中加载
- collate_fn：合并样本列表以形成小批量。
- pin_memory：如果为 True，数据加载器在返回前将张量复制到 CUDA 固定内存中。
- drop_last：如果数据集大小不能被 batch_size 整除，设置为 True 可删除最后一个不完整的批处理。如果设为 False 并且数据集的大小不能被 batch_size 整除，则最后一个 batch 将更小。

首先 DataLoader 是导入图片的操作，里面有一些参数，比如 batch_size 和 shuffle 等，默认 Load 进去的图片类型是 PIL.Image.open 的类型。

```
    #加载训练集
    test_loader = 
torch.utils.data.DataLoader(dataset=test_dataset,
                                                 batch_size=batch_size,
                                                 shuffle=False)
    #加载测试集
    Model
    class LogisticRegression(nn.Module):
        def __init__(self, input_size, num_classes):
            super(LogisticRegression, self).__init__()
            self.linear = nn.Linear(input_size, num_classes)
        #定义逻辑回归模型，这里的 nn.Linear 表示的是 y=w×x+b
        def forward(self, x):
            out = self.linear(x)
            return out

    net=LogisticRegression(5,1)
    print(net)
    #结果如下
    LogisticRegression ((linear): Linear (5 -> 1))

    model = LogisticRegression(input_size, num_classes)
```

定义 Loss 和 Optimizer，就是误差和优化函数。这里使用的是最小二乘 Loss，之后我们做分类问题更多的使用的是 Cross Entropy Loss，交叉熵。优化函数使用的是随机梯度下降。

```
    loss= nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(),
lr=learning_rate)
    #SGD 即随机梯度下降。
```

随机梯度下降算法每次从训练集中随机选择一个样本来进行学习，批量梯度下降算法每次都会使用全部训练样本，因此这些计算是冗余的，因为每次都使用完全相同的样本集。而随机梯度下降算法每次只随机选择一个样本来更新模型参数，因此每次的学习是非常快速的，并且可以进行在线更新。因此，其速度较快，但是其每次的优化方向不一定是全局最优的，但最终的结果是在全局最优解的附近。

```
    # 对模型进行训练
    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(train_loader):
            images = Variable(images.view(-1, 28*28))
            labels = Variable(labels)
            #加载的数据变成 pytorch 的 tensor
            optimizer.zero_grad()
    #设置梯度为零
            outputs = model(images)
            loss = loss(outputs, labels)#计算损失
            loss.backward()
            optimizer.step()
    #更新参数


            if (i+1) % 100 == 0:
                print ('Epoch: [%d/%d], Step: [%d/%d], Loss: %.4f'
                        % (epoch+1, num_epochs, i+1,
len(train_dataset)//batch_size, loss.data[0]))
```

输出数据。

```
    # 对模型进行测试，计算模型的精度
    correct = 0
```

```
    total = 0
    for images, labels in test_loader:
        images = Variable(images.view(-1, 28*28))
    outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()
    #打印模型测试的精度
    print('Accuracy of the model on the 10000 test images: %d %%' %
(100 * correct / total))

    # 保存模型
    torch.save(model.state_dict(), 'model.pkl')
```

模型的结果如下。

```
Epoch: [1/10], Step: [100/1200], Loss: 2.2257
Epoch: [1/10], Step: [200/1200], Loss: 2.1514
…………………………………………………….
Epoch: [10/10], Step: [1000/1200], Loss: 0.4795
Epoch: [10/10], Step: [1100/1200], Loss: 0.632
Epoch: [10/10], Step: [1200/1200], Loss: 0.5013
Accuracy of the model on the 10000 test images: 87 %
#经过10批次的训练，可以看出模型的准确率为87%，结果还不错。
```

至此已经介绍了线性回归模型和逻辑回归模型，相信大家已经对 PyTorch 有了一定了解，下一章节开始进入神经网络的学习。

第 5 章 前馈神经网络

常见的前馈神经网络有感知机（Perceptrons）、BP（Back Propagation）网络、RBF（Radial Basis Function）网络等。

感知器（又叫感知机）是最简单的前馈网络，它主要用于模式分类，也可用在基于模式分类的学习控制和多模态控制中。感知器网络可分为单层感知器网络和多层感知器网络。

BP 网络是指连接权调整采用了反向传播（Back Propagation）学习算法的前馈网络。与感知器不同之处在于，BP 网络的神经元变换函数采用了 S 形函数（Sigmoid 函数），因此输出量是 0～1 之间的连续量，可实现从输入到输出的任意的非线性映射。

RBF 网络是指隐含层神经元由 RBF 神经元组成的前馈网络。RBF 神经元是指神经元的变换函数为 RBF（Radial Basis Function，径向基函数）的神经元。典型的 RBF 网络由三层组成：一个输入层，一个或多个由 RBF 神经元组成的 RBF 层（隐含层），一个由线性神经元组成的输出层。

简单的多层感知机，如图 5.1 所示。

图 5.1 中每个圆圈都是一个神经元，每条线表示神经元之间的连接。我们可以看到，上面的神经元被分成了多层，层与层之间的神经元有连接，而层内之间的神经元没有连接。最左边的层叫作输入层，这层负责接收输

入数据；最右边的层叫作输出层，我们可以从这层获取神经网络输出数据。输入层和输出层之间的层叫作隐藏层。神经网络的组成单元是神经元，也叫作感知器。感知器算法在 20 世纪 50 年代到 70 年代很流行，也成功解决了很多问题。并且，感知器算法也是非常简单的。

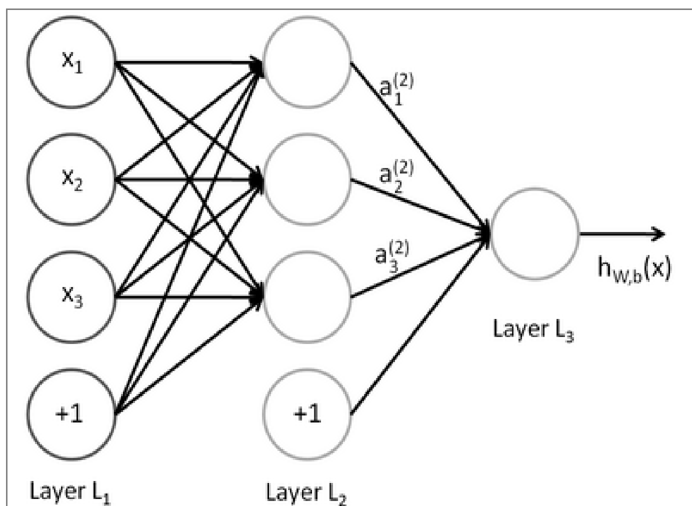


图 5.1　简单的多层感知机

我们知道感知器实际上是神经网络结构中的一个神经元，那么一个感知器就构成了最简单的神经网络。感知器以一个实数值向量作为输入，计算这些输入的线性组合，然后如果结果大于某个阈值就输出 1，否则输出 –1。x_1，x_2 和 x_3 是输入单元。每个输入单元分别代表一个特征。每个输入上有一个权值 w_i，此外还有一个偏置项 w_0。感知器通过使用激励函数（Activation Function）处理解释变量和模型参数的线性组合对样本分类。如果结果大于某个阈值就输出 1，否则输出–1。

感知器计算的输出为：

$$O(x_1,\cdots,x_n)=\begin{cases}1 & \text{if } w_0+w_1x_1+w_2x_2+\cdots+w_nx_n>0\\ -1 & \text{otherwise}\end{cases}$$

其中每一个 w_i 是一个实数常量，或叫作权值（Weight）

下面我们用代码来实现多层感知机。

5.1　实现前馈神经网络

前面已经简单介绍了前馈神经网络的原理，接下来我们用案例来实现吧。

本例文件名 PyTorch/Chapter05/pt01_feedforward_neural_network.py。

```
    import torch
    import torch.nn as nn
    import torchvision.datasets as dsets
    import torchvision.transforms as transforms
    from torch.autograd import Variable
    import torch.utils.data as Data
    import matplotlib.pyplot as plt#导入 Torch 模块，加载 Torchvision，
这是专门做图形处理的一个库，加载 Dsets 数据集，我们需要使用的是
torchvision.transforms、torchvision.datasets 以及
torch.utils.data.DataLoader。

    input_size = 784
    hidden_size = 500
    num_classes = 10
    num_epochs = 5
    batch_size = 100
    learning_rate = 0.001
    #图片大小为 784,图片大小为 784,尺寸为 28×28
    train_dataset = dsets.MNIST(root='./data',
                               train=True,
                               transform=transforms.ToTensor(),
                               download=True)

    test_dataset = dsets.MNIST(root='./data',
                              train=False,
                              transform=transforms.ToTensor())
```

参数说明如下。

- root：数据集，存在于根目录 processed/training.pt 和 processed/test.pt 中。

- train：True＝ 训练集，False＝ 测试集。
- download：如果为 True，请从 Internet 下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。
- transform：接收 PIL 映像并返回转换版本的函数/变换。
- target_transform：一个接收目标并转换它的函数/变换。

```
    train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
                                               batch_size=batch_size,
                                               shuffle=True)


    test_loader =
torch.utils.data.DataLoader(dataset=test_dataset,
                                              batch_size=batch_size,
                                              shuffle=False)
    #加载手写图片数据，torchvision.datasets 里面提供了丰富的类型图片
    train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=batch_size, shuffle=True)
```

参数说明如下。

- dataset：加载数据的数据集。
- batch_size：加载批训练的数据个数。
- shuffle：为 True 则在每个 Epoch 重新排列数据。
- sampler：从数据集中提取样本。
- batch_sampler：一次返回一批索引。
- num_workers：用于数据加载的子进程数。0 表示数据将在主进程中加载。
- collate_fn：合并样本列表以形成小批量。
- pin_memory：如果为 True，数据加载器在返回前将张量复制到 CUDA 固定内存中。
- drop_last：如果数据集大小不能被 batch_size 整除，设置为 True 可删除最后一个不完整的批处理。如果设为 False 并且数据集的大小不能被 batch_size 整除，则最后一个 batch 将更小。

首先 DataLoader 是导入图片的操作，里面有一些参数，比如 batch_size 和 shuffle 等，默认 Load 进去的图片类型是 PIL.Image.open 的类型。

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

```
net = Net(input_size, hidden_size, num_classes)
```

前馈神经网络由一个输入层、一个或多个隐藏层和一个输出层组成，前馈神经网络结构包括：输入层的单元数、隐藏层数（如果多于一层）、每个隐藏层的单元数和输出层的单元数。神经网络可以用于分类（预测给定元组的类标号）和数值预测（预测连续值输出）等。input_size 输入的图片大小为 784，hidden_size 为 500，num_classes 为 10，即输入的数据大小为 784，隐藏层的神经元个数为 500 个，输入的数据经过激励函数的线性变换，输出大小为 10 个神经元。隐含层神经元如果过少，容易产生过多的局部极小，容错性较差。如果隐含层的神经元过多，学习训练时间太长，但不一定为最佳。隐含层的神经元数目多少决定了多层前馈神经网络的可区分性能，同时计算的复杂性由隐含层的神经元多少决定。

```
criterion = nn.CrossEntropyLoss()
```

损失函数（Loss Function）是用来估量模型的预测值 $f(x)$ 与真实值 Y 的不一致程度，它是一个非负实值函数，通常使用 L（Y，$f(x)$）来表示，损失函数越小，模型的鲁棒性就越好。损失函数是经验风险函数的核心部分，也是结构风险函数重要组成部分。

常见的损失函数有：Log 对数损失函数，平方损失函数，指数损失函

数（Adaboost），Hinge 损失函数等。这里使用的是最小二乘 Loss，但是我们做分类问题更多的时候使用的是 Cross Entropy Loss，交叉熵。下面我们列出的常见的损失函数。

```
    torch.nn.functional.cross_entropy(input, target, weight=None, size_average=True)
    #交叉熵损失函数
    torch.nn.functional.kl_div(input, target, size_average=True)
    #KL 散度损失函数
    torch.nn.functional.nll_loss(input, target, weight=None, size_average=True)
    #负的 log likelihood 损失函数
    optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
```

要使用 torch.optim，您必须构造一个 Optimizer 对象。这个对象能保存当前的参数状态，并且基于计算梯度更新参数。要构造一个 Optimizer，你必须给它一个包含参数（必须都是 Variable 对象）并进行优化。然后，可以指定 Optimizer 的参数选项，比如学习率、权重衰减等。Optimizer 也支持为每个参数单独设置选项。我们不需要直接传入 Variable 的 Iterable，而是利用 Dict 进行传入数据。每一个 Dict 都分别定义了一组参数，实现一一对应。其他的参数与 Optimizer 所接收的其他参数的关键字相匹配，实现参数优化。一些优化算法例如 Conjugate Gradient 和 LBFGS 需要重复多次计算函数，因此你需要传入一个闭包来允许它们重新计算你的模型。这个闭包会清空梯度，计算损失，然后返回。常用的优化函数有 SGD 全名 Stochastic Gradient Descent，即随机梯度下降。NesterovMomentum 优化函数等。

```
    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(train_loader):
            images = Variable(images.view(-1, 28*28))
            labels = Variable(labels)
           #对批量训练的图片机型循环，输入的图片大小为 28×28 转变为 Torch 可以识别的变量，图片标签转换成可以识别的标签，然后进行训练


            optimizer.zero_grad()
            outputs = net(images)
```

```
    #把图片的信息传给 outputs
            loss = criterion(outputs, labels)
            loss.backward()
    #计算损失函数
            optimizer.step()
    #更新参数

            if (i+1) % 100 == 0:
                print ('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'
                       %(epoch+1, num_epochs, i+1,
len(train_dataset)//batch_size, loss.data[0]))

    #每 100 进行输出结果
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = Variable(images.view(-1, 28*28))
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()
    #输出精度
    print('Accuracy of the network on the 10000 test images: %d %%'
% (100 * correct / total))
```

输出的结果为 97%，说明前馈神经网络对手写图片的分类具有良好的分类效果。经过 5 个批次 600 次训练，准确率达到 97%。

```
    Epoch [1/5], Step [100/600], Loss: 0.3444
    Epoch [1/5], Step [200/600], Loss: 0.3170
    Epoch [1/5], Step [300/600], Loss: 0.2504
    Epoch [1/5], Step [400/600], Loss: 0.2294
    Epoch [1/5], Step [500/600], Loss: 0.1255
    Epoch [1/5], Step [600/600], Loss: 0.2812
    Epoch [2/5], Step [100/600], Loss: 0.0826
    ……
    Epoch [4/5], Step [600/600], Loss: 0.0635
    Epoch [5/5], Step [100/600], Loss: 0.0561
```

```
    Epoch [5/5], Step [200/600], Loss: 0.0373
    Epoch [5/5], Step [300/600], Loss: 0.0105
    Epoch [5/5], Step [400/600], Loss: 0.0434
    Epoch [5/5], Step [500/600], Loss: 0.0099
    Epoch [5/5], Step [600/600], Loss: 0.0602
    Accuracy of the network on the 10000 test images: 97 %
    for i in range(1,4):

        plt.imshow(train_dataset.train_data[i].numpy(),
cmap='gray')

        plt.title('%i' % train_dataset.train_labels[i])

    plt.show()
```

我们输出图片看看据图的手写图片的效果。

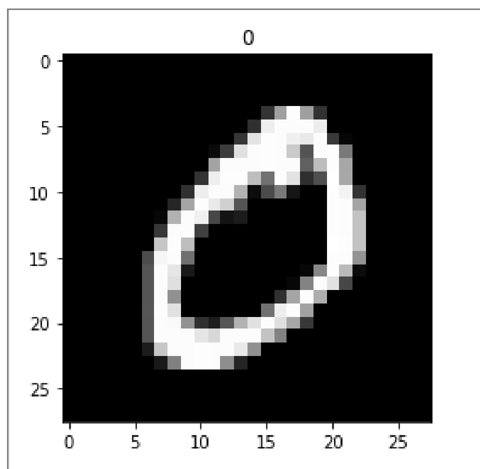


图 5.2　0 的手写图片

输出第一张照片是 0 的手写照片，对应的标签也是 0，如图 5.2 所示。

输出第二张照片是数字 4 的手写照片，对应的标签也是 4，如图 5.3 所示。

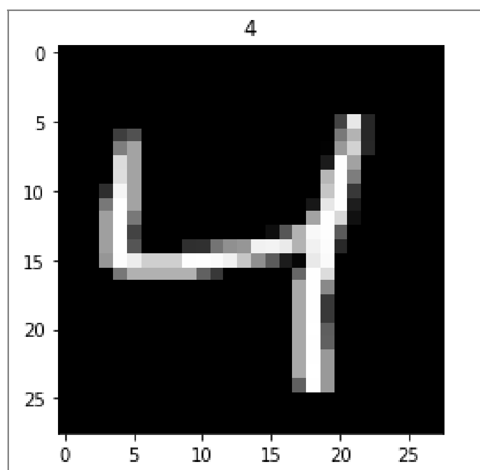


图 5.3　4 的手写图片

输出第三张照片是数字 1 的手写照片，对应的标签也是 1，如图 5.4 所示。

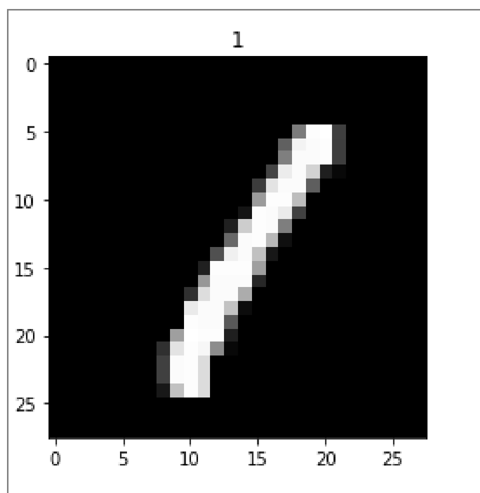


图 5.4　1 的手写图片

```
test_output = net(images[:20])
pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
print('prediction number',pred_y)
print('real number',test_y[:20].numpy())
```

输出前 10 张照片的真实值与预测值，从对比的结果发现可以看出，单层神经网络预测具有良好的效果。

```
prediction number[7,2,1,0,4,1,4,9,5,9]
real number[7,2,1,0,4,0,4,5,2,1]
#保存模型
torch.save(net.state_dict(), 'model.pkl')
```

我们从前馈神经网络模型的案例中已经大概了解 PyTorch 形成神经网络的流程。下面我们来具体介绍一下，形成完整的神经网络模型所需的 PyTorch 包的组成。

5.2 数据集

Torchvision 包括了目前流行的数据集、模型结构和常用的图片转换工具。

torchvision.datasets 中包含了以下数据集：

- MNIST
- COCO（用于图像标注和目标检测）（Captioning and Detection）
- LSUN Classification
- ImageFolder
- Imagenet-12
- CIFAR10 and CIFAR100
- STL10
- SVHN
- PhotoTour

MNIST 介绍如下：

dset.MNIST(root,train=True,transform=None,target_transform=None, download=False)

参数说明。

- root：数据集，存在于根目录 processed/training.pt 和 processed/test.pt 中。
- train：True = 训练集，False = 测试集。

- download：如果为 True，请从 Internet 下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。
- transform：接收 PIL 映像并返回转换版本的函数/变换。
- target_transform：一个接收目标并转换它的函数/变换。

加载数据如下：

torch.utils.data.DataLoader(dataset,batch_size=1,shuffle=False,sampler=None,num_workers=0,collate_fn=<function default_collate>,pin_memory=False,drop_last=False)

参数如下。

- Dataset：加载数据的数据集。
- batch_size：加载批训练的数据个数。
- Shuffle：如果为 True，在每个 epoch 重新排列数据。
- Sampler：从数据集中提取样本。
- batch_sampler：一次返回一批索引。
- num_workers：用于数据加载的子进程数。0 表示数据将在主进程中加载
- collate_fn：合并样本列表以形成小批量。
- pin_memory：如果为 True，数据加载器在返回前将张量复制到 CUDA 固定内存中。
- drop_last：如果数据集大小不能被 batch_size 整除，设置为 True 可删除最后一个不完整的批处理。如果设置为 False 并且数据集的大小不能被 batch_size 整除，则最后一个 batch 将更小。

torchvision.models 模块的子模块中包含以下预训练的模型结构。

- AlexNet
- VGG
- ResNet
- SqueezeNet
- DenseNet

ResNet（Residual Neural Network）是由微软研究院的 Kaiming He 等 4

名华人提出的，一般我们把 ResNet 称作是残差网。在 ILSVRC 2015 的比赛中，Kaiming He 等人因为通过使用 Residual Units（残差单元）训练了 152 层神经网络而赢得了冠军。在比赛中的 Top-5 的错误率为 3.57%，同时参数量比 VGGNet 低，效果非常突出。ResNet 在 2015 年名声大噪，而且影响了 2016 年深度学习在学术界和工业界的发展方向。

ResNet 引入了残差网络结构（Residual Network），通过残差网络，可以把网络层弄得很深。利用 ResNet 强大的表征能力，不仅是图像分类，而且很多其他计算机视觉应用（比如物体检测和面部识别）的性能都得到了极大的提升。

VGGNet，牛津大学计算机视觉组（Visual Geometry Group）和 Google DeepMind 公司一起研发，为深度卷积神经网络。VGGNet 反复堆叠 3×3 小型卷积核和 2x2 最大池化层，成功构筑 16～19 层深卷积神经网络。相比最新的网络结构，错误率大幅下降，取得 ILSVRC 2014 比赛分类第 2 名和定位第 1 名的成绩。VGGNet 拓展性强，迁移其他图片数据泛化性好。VGGNet 模型的准确率相比于 AlexNet 有了很大提升，VGGNet 虽然模型参数比 AlexNet 多，但反而只需要较少的迭代次数就可以收敛，主要原因是更深的网络和更小的卷积核带来的隐式的正则化效果。VGGNet 凭借其相对不算很高的复杂度和优秀的分类性能，一代经典的卷积神经网络，直到现在依然被应用在很多地方。

可以通过调用构造函数来构造具有随机权重的模型：

```
import torchvision.models as models
resnet18 = models.resnet18()
alexnet = models.alexnet()
squeezenet = models.squeezenet1_0()
densenet = models.densenet_161()
```

PyTorch 提供了大量的预训练的模型，利用 PyTorch 的 torch.utils.model_zoo 来加载预训练模型。这些可以通过构建 pretrained=True，如我们加载预训练的 resnet18 模型。

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
```

下面是 PyTorch 提供的预训练的模型。

```
torchvision.models.alexnet(pretrained=False, ** kwargs)
```

构建一个 AlexNet 模型结构。

Torch.nn 包里面包含了如何定义 Module 神经网络模型，各种 Functional 函数以及如何通过 Optim 实现各种优化算法。Module 是神经网络的基本组成部分，作为一个抽象类，可以通过定义成员函数实现不同的神经网络结构。Functional 包括 Convolution 函数、Pooling 函数、非线性激活函数、Normalization 函数、线性函数、Dropout 函数、距离函数（Distance functions）、损失函数（Loss functions）等。Optim 包含了各种优化算法，Momentum 算法、NesterovMomentum 算法、AdaGrad 算法、RMSProp 算法、Adam（Adaptive Moment Estimation）算法等，我们可以根据神经网络模型的需求选择合适的优化算法。

本例文件名为 PyTorch/Chapter05/pt02_model.py。

我们先来看看如何定义神经网络模型的架构。

```
class torch.nn.Module
```

Modules 还可以包含其他模块，允许将它们嵌套在树结构中。可以将子模块分配为常规属性。

```
import torch.nn as nn
import torch.nn.functional as F
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
       x = F.relu(self.conv1(x))
       return F.relu(self.conv2(x))
```

以这种方式分配的子模块将被注册，并且在调用.cuda()等时也会转换参数。

5.3 卷积层

卷积是一种局部操作，通过一定大小的卷积核作用于局部图象区域，从而得到图象的局部信息。下面以一维卷积层为例：

class torch.nn.Conv1d(in_channels,out_channels,kernel_size,stride=1, padding=0,dilation=1,groups=1,bias=True)

一维卷积层，输入的尺度是（N,C_in,L），输出尺度（N,C_out,L_out）

参数说明如下。

- in_channels（int）：输入信号的通道。
- out_channels（int）：卷积产生的通道。
- kerner_size（int or tuple）：卷积核的尺寸。
- stride（int or tuple，optional）：卷积步长。
- padding（int or tuple，optional）：是否对输入数据填充 0。Padding 可以将输入数据的区域改造成是卷积核大小的整数倍，这样对不满足卷积核大小的部分数据就不会忽略了。通过 padding 参数指定填充区域的高度和宽度。
- dilation（int or tuple，`optional`）：卷积核之间的空格。
- groups（int，optional）：将输入数据分成组，in_channels 应该被组数整除。
- bias（bool，optional）：如果 bias=True，添加偏置。

例子：

```
>>> m = nn.Conv1d(16, 33, 3, stride=2)
>>> inputs = autograd.Variable(torch.randn(20, 16, 50))
>>> output = m(inputs)
```

池化层如下：

class torch.nn.MaxPool1d(kernel_size,stride=None,padding=0,dilation=1, return_indices=False,ceil_mode=False)

对于输入信号的输入通道，提供一最大池化（Max Pooling）操作。

参数说明如下。

- kernel_size（int or tuple）：Max Pooling 的窗口大小。
- stride（int or tuple，optional）：max pooling 的窗口移动的步长。默认值是 kernel_size。
- padding（int or tuple，optional）：输入的每一条边补充 0 的层数。
- dilation（int or tuple，optional）：一个控制窗口中元素步幅的参数。
- return_indices：如果等于 True，会返回输出最大值的序号，对于上采样操作会有帮助
- ceil_mode：如果等于 True，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作。

例子：

```
>>> m = nn.MaxPool1d(3, stride=2)
>>> input = autograd.Variable(torch.randn(20, 16, 50))
>>> output = m(input)
```

我们用 nn 包来定义我们的神经网络。如果要创建一个经常性网络，只需多次使用相同的线性图层，而无需考虑分享权重。这样让我们在使用神经网络时变得更简单、更高效。

下面我们创建一个小型 ConvNet。所有的网络都是从基类 nn.Module 中构造函数。本例文件名为 PyTorch/Chapter05/pt03_nn.py。

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
class MNISTConvNet(nn.Module):
    def __init__(self):
        super(MNISTConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(10, 20, 5)
```

```
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)
    def forward(self, input):
        x = self.pool1(F.relu(self.conv1(input)))
        x = self.pool2(F.relu(self.conv2(x)))
        return x
net = MNISTConvNet()
print(net)
```

输出结果如下。

```
MNISTConvNet (
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1,
1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1,
1))
  (fc1): Linear (320 -> 50)
  (fc2): Linear (50 -> 10)
)
```

torch.nn 包装仅支持作为小批量样品的输入，而不是单个样品。

```
input = Variable(torch.randn(1, 1, 28, 28))
out = net(input)
print(out.size())
```

输出结果如下。

```
torch.Size([1, 10])
```

定义虚拟目标标签并使用损失函数计算错误。

```
target = Variable(torch.LongTensor([3]))
loss_fn = nn.CrossEntropyLoss()
err = loss_fn(out, target)
err.backward()
print(err)
```

输出结果如下。

```
Variable containing:
 2.3186
[torch.FloatTensor of size 1]
```

让我们访问各个层的权重和梯度：

```
print(net.conv1.weight.grad.size())
```

输出结果如下。

```
torch.Size([10, 1, 5, 5])

print(net.conv1.weight.data.norm())
print(net.conv1.weight.grad.data.norm())
```

输出结果如下。

```
1.8083854303685114
0.1320870710384528
```

5.4　Functional函数

torch.nn.functional 包里面提供的函数如下。

- Convolution 函数
- Pooling 函数
- 非线性激活函数
- Normalization 函数
- 线性函数
- Dropout 函数
- 距离函数（Distance Functions）
- 损失函数（Loss Functions）
- Vision Functions

nn.functional 中的函数仅仅定义了一些具体的基本操作，不能构成 PyTorch 中的一个 Layer。当你需要自定义一些非标准 Layer 时，可以在其中调用 nn.functional 中的操作。比如 F.relu 仅仅是一个函数，参数包括输入和计算所需参数，返回计算结果，它不能存储任何上下文信息。所有的

Function 函数都从基础类 Function 派生，实现 Forward 和 Backward 静态方法。而在 Forward 和 Backward 实现内部，调用了 C 的后端实现。

Torch.nn 包里面只是包装好了神经网络架构的类，nn.functional 与 Torch.nn 包相比，nn.functional 是可以直接调用函数的。

我们来具体看一看下面这个函数：

torch.nn.functional.conv1d(input,weight,bias=None,stride=1,padding=0, dilation=1,groups=1)

对输入的数据进行一维卷积。卷积的本质就是用卷积核的参数来提取原始数据的特征，通过矩阵点乘的运算，提取出和卷积核特征一致的值，如果卷积层有多个卷积核，则神经网络会自动学习卷积核的参数值，使每个卷积核可以代表一个特征。

参数说明如下。

- input 输入的 Tensor 数据，格式为（batch，channels，W），三维数组，第一维度是样本数量，第二维度是通道数或者记录数，第三维度是宽度。
- weight：过滤器，也叫卷积核权重。是一个三维数组，（out_channels，in_channels/groups，kW）。out_channels 是卷积核输出层的神经元个数，也就是这层有多少个卷积核；in_channels 是输入通道数；kW 是卷积核的宽度。
- bias：位移参数。
- stride：滑动窗口，默认为 1，指每次卷积对原数据滑动 1 个单元格。
- padding：是否对输入数据填充 0。Padding 可以将输入数据的区域改造成是卷积核大小的整数倍，这样对不满足卷积核大小的部分数据就不会忽略了。通过 Padding 参数指定填充区域的高度和宽度，默认为 0。
- dilation：卷积核之间的空格，默认为 1。
- groups：将输入数据分成组，in_channels 应该被组数整除，默认为 1。
- conv1d 是一维卷积，它和 conv2d 的区别在于只对宽度进行卷积，对高度不卷积。

例子：

```
>>> filters = autograd.Variable(torch.randn(33, 16, 3))
>>> inputs = autograd.Variable(torch.randn(20, 16, 50))
>>> F.conv1d(inputs, filters)
```

1．Pooling 函数

Pooling 函数主要是用于图像处理的卷积神经网络中，但随着深层神经网络的发展，Pooling 函数相关技术在其他领域，其他结构的神经网络中也越来越受关注。

卷积神经网络中的卷积层是对图像的一个邻域进行卷积得到图像的邻域特征。亚采样层就是使用 Pooling 函数技术将小邻域内的特征点整合得到新的特征。Pooling 函数确实起到了整合特征的作用。

池化操作是利用一个矩阵窗口在张量上进行扫描，将每个矩阵通过取最大值或者平均值等方法来减少元素的个数，最大值和平均值的方法可以使得特征提取拥有“平移不变性”，也就说图像有了几个像素的位移情况下，依然可以获得稳定的特征组合，平移不变性对于识别十分重要。

Pooling 函数的结果是特征减少，参数减少，但 Pooling 的目的并不仅在于此。Pooling 函数的目的是保持某种不变性（旋转、平移、伸缩等），常用的有 Mean-Pooling 函数，Max-Pooling 函数和 Stochastic-Pooling 函数三种。我们以一维平均池化为例进行说明：

torch.nn.functional.avg_pool1d（input，kernel_size，stride=None，padding=0，ceil_mode=False，count_include_pad=True）

对由几个输入平面组成的输入信号进行一维平均池化。avg_pool1d，即对邻域内特征点只求平均：假设 Pooling 的窗大小是 2×2，在 Forward 的时候，就是在前面卷积完的输出上依次不重合地取 2×2 的窗平均，得到一个值就是当前 avg_pool1d 之后的值。

参数说明如下。

- kernel_size：窗口的大小。
- stride：窗口的步长，默认值为 kernel_size。

- padding：是否对输入数据填充 0。Padding 可以将输入数据的区域改造成是卷积核大小的整数倍，这样对不满足卷积核大小的部分数据就不会忽略了。通过 Padding 参数指定填充区域的高度和宽度。
- ceil_mode：当为 True 时，将使用 Ceil 代替 Floor 来计算输出形状。
- count_include_pad：当为 True 时，将包括平均计算中的 0 填充。默认为 True。

例子：

```
>>> input = Variable(torch.Tensor([[[1,2,3,4,5,6,7]]]))
>>> F.avg_pool1d(input, kernel_size=3, stride=2)
Variable containing:
(0 ,.,.) =
  2  4  6
[torch.FloatTensor of size 1x1x3]
```

Sigmoid 函数，Sigmoid 函数曾被广泛地应用，但由于其自身的一些缺陷，现在很少被使用了。具体如下：

torch.nn.functional.sigmoid（input）

Sigmoid 函数被定义为：$f(x)=\dfrac{1}{1+\mathrm{e}^{-x}}$

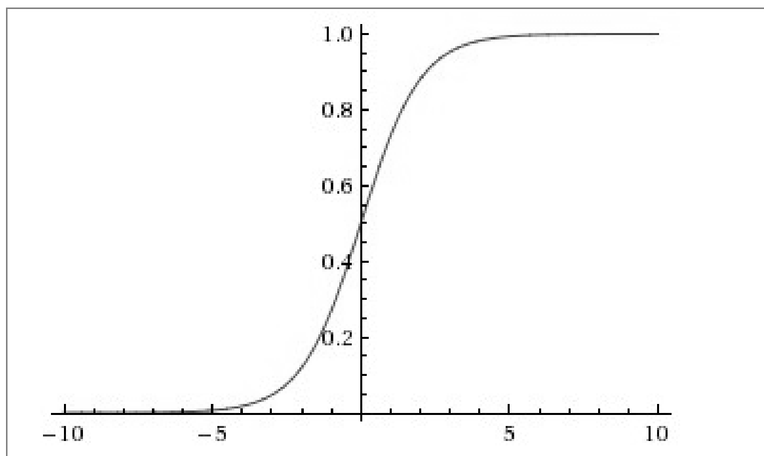


图 5.4　Sigmoid 函数

优点：

① Sigmoid 函数的输出映射在（0，1）之间，单调连续，输出范围有限，优化稳定，可以用作输出层；

② 求导容易。

缺点：

① 由于其软饱和性，容易产生梯度消失，导致训练出现问题；

② 其输出并不是以 0 为中心的。

```
>>> m = nn.sigmoid()
```

2. Tanh 函数

Tanh 函数是双曲函数中的一个，为反曲正切。具体如下：

torch.nn.functional.tanh(input)

Tanh 函数被定义为： $\tanh(x) = \dfrac{1-\mathrm{e}^{-2x}}{1+\mathrm{e}^{-2x}}$

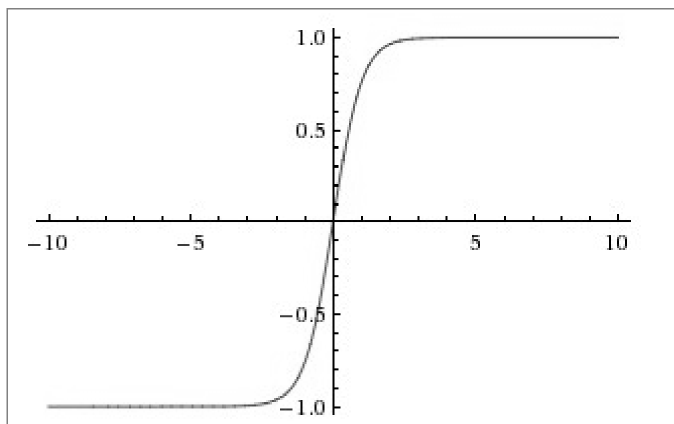


图 5.5　Tanh 函数

优点：

① 比 Sigmoid 函数收敛速度更快；

② 相比 Sigmoid 函数，其输出以 0 为中心。

缺点：

还是没有改变 Sigmoid 函数的最大问题——由于饱和性产生的梯度消失。

```
>>> m = nn.tanh()
```

3．ReLU 函数

ReLU 函数用来替代传统的激活函数，如下所示：

torch.nn.functional.relu(input,inplace=False)

ReLU 函数被定义为：$y=\begin{cases}0 & (x\le 0)\\ x & (x>0)\end{cases}$

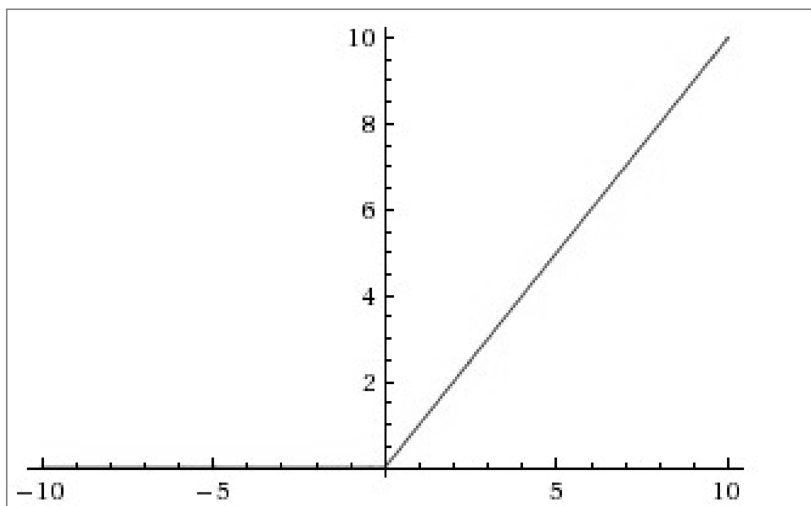


图 5.6　ReLU 函数

```
>>> m = nn.ReLU()
```

4．Dropout 函数

Dropout 函数是 Hintion 在 2012 年提出的。为了防止模型过拟合，Dropout 可以作为一种 Trikc 供选择。在每个训练批次中，通过忽略一半的神经元即让一半的隐层节点值为 0，可以明显地减少过拟合现象。下面我们列出常见的 Dropout 函数：

- torch.nn.functional.dropout(input,p=0.5,training=False,inplace=False)
- torch.nn.functional.alpha_dropout(input,p=0.5,training=False)
- torch.nn.functional.dropout2d(input，p=0.5,training=False,inplace=False)
- torch.nn.functional.dropout3d(input，p=0.5,training=False,inplace=False)

5．损失函数（Loss functions）

通常机器学习每一个算法中都会有一个目标函数，算法的求解过程是通过对这个目标函数优化的过程。在分类或者回归问题中，通常使用损失函数（代价函数）作为其目标函数。损失函数用来评价模型的预测值和真实值不一样的程度，损失函数越好，通常模型的性能越好。不同的算法使用的损失函数不一样。

损失函数分为经验风险损失函数和结构风险损失函数。经验风险损失函数指预测结果和实际结果的差别，结构风险损失函数是指经验风险损失函数加上正则项。下面我们列出常见的损失函数。

KL 散度损失函数如下：

torch.nn.functional.kl_div(input,target,size_average=True)

参数如下：

- input：变量的任意形状。
- target：与输入相同形状的变量。
- size_average：如果是真的，输出就除以输入张量中的元素个数。

其他常见的损失函数如下：

torch.nn.functional.poisson_nll_loss(input,target,log_input=True,full=False, size_average=True)

torch.nn.functional.nll_loss(input,target,weight=None,size_average=True)

torch.nn.functional.cosine_embedding_loss(input,input2,target,margin=0, size_average=True)

5.5 优化算法

torch.optim 是实现各种优化算法的包。

在使用 torch.optim 包构建 Optimizer 对象中，可以指定 Optimizer 参数，包括学习速率，权重衰减等。

例如：

```
    optimizer = optim.SGD(model.parameters(), lr = 0.01,
momentum=0.9)
    optimizer = optim.Adam([var1, var2], lr = 0.0001)
```

同时也可以为每个参数单独设置选项，利用 dict 定义一组参数，进行传入数据。

下面我们来看一个例子。

当我们想指定每一层的学习速率时，可以使用下面的方法：

```
    optim.SGD([
                {'params': model.base.parameters()},
                {'params': model.classifier.parameters(), 'lr':
1e-3}
                ], lr=1e-2, momentum=0.9)
```

model.base 参数将使用默认的学习速率 1^{e-2}，model.classifier 参数将使用学习速率 1^{e-3}，并且 0.9 的 momentum 将会被用于所有的参数。

所有的 Optimizer 都会实现 step()更新参数的方法，使用方法如下：

optimizer.step()

一旦梯度被如 backward()之类的函数计算好后，我们就可以调用该函数。

例子如下。

```
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
```

```
    loss.backward()
    optimizer.step()
optimizer.step(closure)
```

一些优化算法例如 Conjugate Gradient 和 LBFGS 需要重复多次计算函数，因此你需要传入一个闭包来允许它们重新计算你的模型。这个闭包会清空梯度，计算损失，然后返回。

例子如下。

```
for input, target in dataset:
    def closure():
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        return loss
optimizer.step(closure)
```

下面介绍常见的随机梯度优化算法：

torch.optim.SGD(params,lr=,momentum=0,dampening=0,weight_decay=0, nesterov=False)

以上可以实现随机梯度下降算法（momentum 可选）。

SGD 全名 Stochastic Gradient Descent，即随机梯度下降。其特点是训练速度快，对于很大的数据集，也能够以较快的速度收敛。

由于是抽取，因此得到的梯度肯定有误差。因此学习速率需要逐渐减小，否则模型无法收敛。因为误差，所以每一次迭代的梯度受抽样的影响比较大，也就是说梯度含有比较大的噪声，不能很好地反映真实梯度。

选择合适的学习速率比较困难，所以对所有的参数更新使用同样的学习速率。对于稀疏数据或者特征，有时我们可能，所以对于不经常出现的特征，对于常出现的特征更新慢一些，这时候 SGD 就不太能满足要求了。

参数说明如下。

- params（iterable）：用于优化，可以迭代参数或定义参数组。
- lr（float）：学习速率。

- momentum（float，可选）：动量因子（默认：0）。
- weight_decay（float，可选）：权重衰减（L2 范数）（默认：0）。
- dampening（float，可选）：动量的抑制因子（默认：0）。
- nesterov（bool，可选）：使用 Nesterov 动量（默认：False）。

例子如下。

```
    >>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1,
momentum=0.9)
    >>> optimizer.zero_grad()
    >>> loss_fn(model(input), target).backward()
    >>> optimizer.step()
```

torch.optim.lr_scheduler 提供了几种方法来根据 epoches 的数量调整学习速率。torch.optim.lr_scheduler.ReduceLROnPlateau 允许基于一些验证测量来降低动态学习速率。具体函数如下：

torch.optim.lr_scheduler.LambdaLR(optimizer,lr_lambda,last_epoch=–1)

将每个参数组的学习速率设置为初始的 lr 乘以一个给定的函数。当 last_epoch=-1 时，将初始 lr 设置为 lr。

参数说明如下。

- optimizer（Optimizer）：包装的优化器。
- lr_lambda（function or list）：一个函数来计算一个乘法因子，给定一个整数参数的 epoch，或列表等功能，为每个组 optimizer.param_groups。
- last_epoch（int）：最后一个时期的索引。默认：–1。

例子：

```
    >>> lambda1 = lambda epoch: epoch // 30
    >>> lambda2 = lambda epoch: 0.95 ** epoch
    >>> scheduler = LambdaLR(optimizer, lr_lambda=[lambda1,
lambda2])
    >>> for epoch in range(100):
    >>>     scheduler.step()
    >>>     train(...)
    >>>     validate(...)
```

我们在训练神经网络的时候其他常见的问题。

1．梯度消失与梯度爆炸

深度神经网络训练的时候，采用反向传播方式，该方式背后其实是链式求导，计算每层梯度的时候会涉及一些连乘操作，因此如果网络过深，那么如果连乘的因子大部分小于 1，最后乘积可能趋于 0；另一方面，如果连乘的因子大部分大于 1，最后乘积可能趋于无穷。这就是所谓的梯度消失与梯度爆炸。

2．梯度消失问题

由于反向传播使用链式规则来计算梯度（通过微分），朝向 n 层神经网络的“前”（输入）层将使其修改的梯度以一个较小的值乘以 n 次方，然后再更新之前的固定值。这意味着梯度将指数性减小。n 越大，网络将需要越来越多的时间来有效地训练。

5.6　自动求导机制

每个变量都有两个标志：requires_grad 和 volatile。

1．requires_grad

如果一个变量定义 requires_grad 为 True，后续的这个变量的所有操作可以使用 requires_grad，如果一个变量定义 requires_grad 为 False，变量不需要梯度，在子图中从不执行向后计算。

例子如下。

```
x = Variable(torch.randn(5, 5))
y = Variable(torch.randn(5, 5))
z = Variable(torch.randn(5, 5), requires_grad=True)
a = x + y
a.requires_grad
# False
b = a + z
b.requires_grad
# True
```

当我们在用已经训练好的模型进行训练的时候，如果想要冻结已经训练好的模型参数，我们只需要使用 requires_grad = False 即可冻结参数。例如，下面的例子是我们想要冻结已经预先训练的 resnet18 网络参数，并且对 resnet18 参数进行优化，此时只要切换冻结模型中的 requires_grad 标志就可以了，此时已经预训练的参数就冻结了。

```
    model = torchvision.models.resnet18(pretrained=True)
    for param in model.parameters():
        param.requires_grad = False
    model.fc = nn.Linear(512, 100)
    optimizer = optim.SGD(model.fc.parameters(), lr=1e-2,
momentum=0.9)
```

2．volatile

requires_grad 为 False，变量不需要梯度，在子图中从不执行向后计算。volatile 代表 require_grad 为 False。我们在不进行执行计算，不调用.backward()的时候，volatile 参数特别有用。它将使用绝对最小的内存来评估模型。

volatile 不同于 require_grad 的传递。如果一个变量定义了 volatile 的操作，那么它的输出也将是 volatile。

例子如下。

```
regular_input = Variable(torch.randn(5, 5))
volatile_input = Variable(torch.randn(5, 5), volatile=True)
model = torchvision.models.resnet18(pretrained=True)
model(regular_input).requires_grad
# True
model(volatile_input).requires_grad
# False
model(volatile_input).volatile
# True
model(volatile_input).creator is None
# True
```

5.7　保存和加载模型

序列化和恢复模型有两种主要方法。

第一种：只保存和加载模型参数。

```
torch.save(the_model.state_dict(), PATH)
```

然后进行下一步。

```
the_model = TheModelClass(*args, **kwargs)
the_model.load_state_dict(torch.load(PATH))
```

第二种：保存和加载整个模型。

```
torch.save(the_model, PATH)
```

然后进行下一步。

```
the_model = torch.load(PATH)
```

5.8　GPU加速运算

该包增加了对CUDA张量类型的支持，实现了与CPU张量相同的功能，但使用GPU进行计算。

```
Torch.cuda.current_device()
```

返回当前所选设备的索引。

```
torch.cuda.device（idx）
```

更改所选设备。

参数说明如下。

- idx（int）：设备索引选择。如果这个参数是负的，则是无效操作。

例子如下。

```
x = torch.cuda.FloatTensor(1)
y = torch.FloatTensor(1).cuda()
```

```
with torch.cuda.device(1):
    a = torch.cuda.FloatTensor(1)
    b = torch.FloatTensor(1).cuda()
    c = a + b
    z = x + y
    d = torch.randn(2).cuda(2)
```

第 6 章

PyTorch 可视化工具

6.1 Visdom介绍

Visdom 是一个灵活的可视化工具（如图 6.1 所示），可以实时显示新创建的数据，支持 Torch 和 Numpy。Visdom 的 github 地址如下。

https://github.com/facebookresearch/visdom

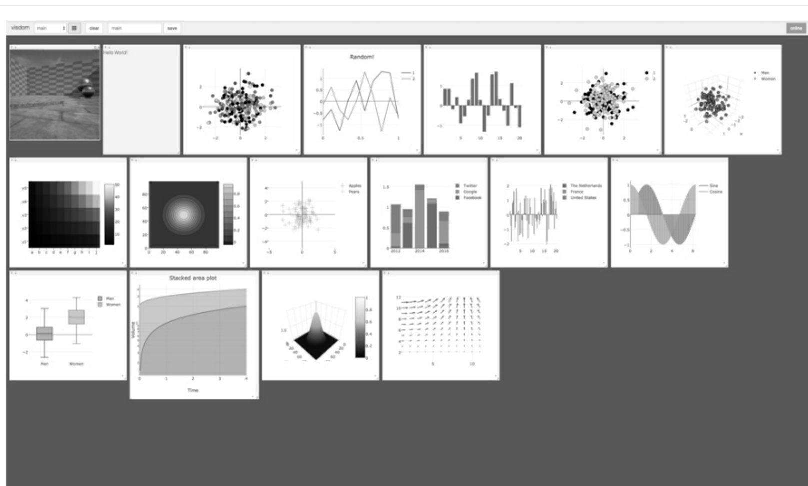


图 6.1　Visdom 可视化工具

Visdom 的目的是促进远程数据的可视化，支持科学实验。可以发送可视化图像和文本。通过 UI 为实时数据创建 dashboards，检查实验的结果。

6.2　Visdom基本概念

Visdom 有一组简单的特性，可以用它们组合成不同的用例。

6.2.1　Panes（窗格）

UI 刚开始是个白板，你可以用图像和文本填充它。这些填充的数据出现在 Panes 中，你可以对这些 Panes 进行拖放、删除、调整大小和销毁操作。Panes 是保存在 Environments（环境）中的，Environments（环境）的状态存储在会话之间。你可以下载 Panes 中的内容，包括你在 SVG 中的绘图。

可以使用浏览器的放大缩小功能来调整 UI 的大小。

6.2.2　Environments（环境）

可以使用 Envs 对可视化空间进行分区。每个用户都会有一个叫作 main 的 Envs。可以通过编程或 UI 创建新的 Envs。Envs 的状态是长期保存的。

可以通过 http://localhost.com:8097/env/main 访问特定的 ENV。如果服务器是被托管的，那么可以将此 URL 分享给其他人，那么其他人也会看到可视化结果。

在初始化服务器的时候，Envs 默认通过$HOME/.visdom/加载。也可以将自定义的路径当作命令行参数传入。如果移除了 Envs 文件夹下的.json 文件，那么相应的环境也会被删除。

6.2.3　State（状态）

一旦你创建了一些可视化，状态是被保存的。服务器自动缓存你的可视化，如果你重新加载网页，你的可视化就会重新出现。

- Save：你可以通过点击 save 按钮手动保存 Envs。它首先会序列化 Envs 的状态，然后以.json 文件的形式保存到硬盘上，包括窗口的位置。同样，你也可以通过编程来实现 Envs 的保存。例如数据丰富的演示、模型的训练仪表板，或者系统实验。这种设计依旧可以使这些可视化十分容易分享和复用。
- Fork：输入一个新的 Envs 名字，“保存”会建立一个新的 Envs：有效地分割之前的状态。

6.3　安装Visdom

在 Python 3 环境下，使用安装 Visdom 模块。

```
pip install visdom
```

启动服务器。

```
python -m visdom.server
```

一旦启动服务器，就可以通过在浏览器中输入 http://localhost:8097 来访问 Visdom，localhost 可以换成 Visdom 服务的托管地址。

6.4　可视化接口

Visdom 支持下列 API。由 Plotly 提供可视化支持。

- vis.text：文本
- vis.image：图片
- vis.scatter：2D 或 3D 散点图
- vis.line：线图
- vis.stem：茎叶图

- vis.heatmap：热力图
- vis.bar：条形图
- vis.histogram：直方图
- vis.boxplot：箱型图
- vis.surf：表面图
- vis.contour：轮廓图
- vis.mesh：网格图
- vis.svg：SVG 图像

这些 API 的确切输入类型有所不同，尽管大多数 API 的输入包含一个 tensor *X*（保存数据）和一个可选的 tensor ***Y***（保存标签或者时间戳）。所有的绘图函数都接收一个可选参数 win，用来将图画到一个特定的 Panes 上。每个绘图函数也会返回当前绘图的 win。你也可以指定汇出的图添加到某个 Envs 上。

6.4.1 Python 函数属性提取技巧

笔者向大家介绍 Python 的原生命令 help，来查看 Visdom 模块下函数的参数与属性，使用脚本如下。

```
from visdom import Visdom

vis= Visdom()
print(help(vis.text ))
```

Visdom 其它绘图的内置函数，也可以使用这种方法查看。输出信息如下。

```
Help on method text in module visdom:

text(text, win=None, env=None, opts=None) method of
visdom.Visdom instance
    This function prints text in a box. It takes as input an `text`
string.
    No specific `opts` are currently supported.
```

6.4.2　vis.text

此函数可在文本框中打印文本。

本案例文件名为 PyTorch/Chapter06/pt06_vm01_text.py，在文本框内输入一个 text 字符串。

```
from visdom import Visdom
vis= Visdom()
vis.text('Hello, world !')
```

运行脚本，显示效果如图 6.2 所示。

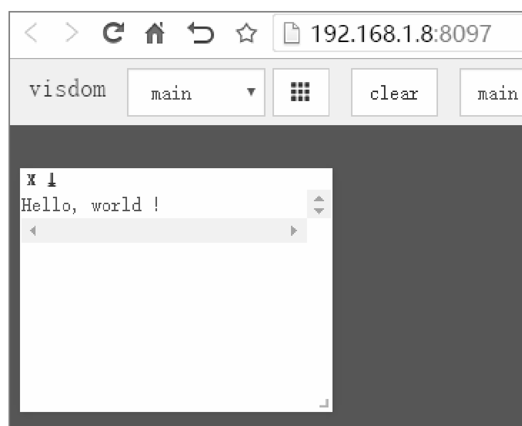


图 6.2　显示文本

6.4.3　vis.image

这个函数用来画图片。本案例文件名为 PyTorch/Chapter06/pt06_vm02_img.py，在页面演示显示图片。

```
from visdom import Visdom

vis= Visdom()

# 显示单图片
vis.image(
```

```
    np.random.rand(3, 256, 256),
    opts=dict(title='单图片', caption='图片标题1'),
)

# 显示网格图片
vis.images(
    np.random.randn(20, 3, 64, 64),
    opts=dict(title='网格图像', caption='图片标题2')
)
```

运行脚本，显示效果如图 6.3 所示。

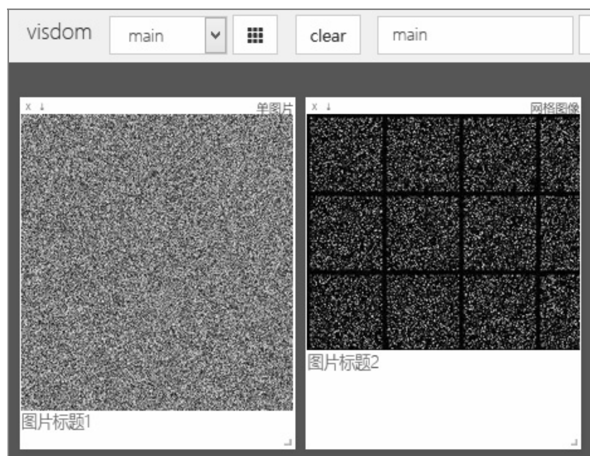


图 6.3　显示图片

6.4.4　vis.scatter

这个函数是用来画 2D 或 3D 数据的散点图。它需要输入 $N\times2$ 或 $N\times3$ 的 tensor $\boldsymbol{X}$ 来指定 N 个点的位置。标签可以通过点的颜色反映出来。

本案例文件名为 PyTorch/Chapter06/pt06_vm03_scatter.py，函数原型如下。

```
from visdom import Visdom
import numpy as np
```

```
vis = Visdom()

# 2D scatterplot with custom intensities (red channel)
vis.scatter(
    X =  np.random.rand(255, 2),
    Y = (np.random.randn(255) > 0) + 1 ,
   opts=dict(
       markersize=10,
       markercolor=np.floor(np.random.random((2, 3)) * 255),
    legend=['Men', 'Women']
    ),
)
```

运行脚本，显示效果如图 6.4 所示。

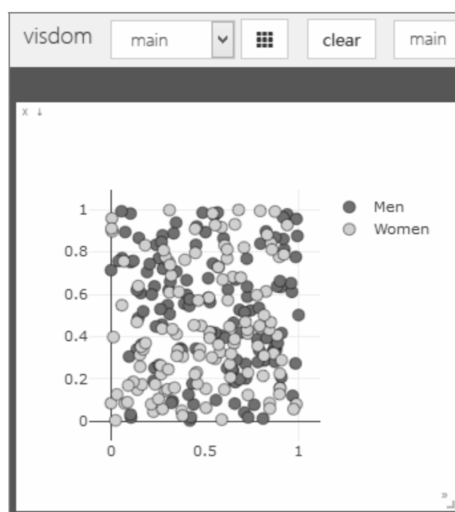


图 6.4　散点图

6.4.5　vis.line

这个函数用来画线图。它需要一个形状为 N 或者 $N\times M$ 的 tensor $\boldsymbol{Y}$，用来指定 M 条线的值（每条线上有 N 个点）。还需要一个可供选择的 tensor $\boldsymbol{X}$ 用来指定对应的 X 轴的值；X 可以是一个长度为 N 的 tensor（这种情况下，M 条线共享同一个 X 轴），也可以是形状和 $\boldsymbol{Y}$ 一样的 tensor。

支持以下属性。

- options.fillarea：填充线下面的区域（boolean）
- options.colormap：色图（string；default = \'Viridis\'）
- options.markers：显示点标记（boolean；default = false）
- options.markersymbol：标记的形状（string；default = \'dot\'）
- options.markersize：标记的大小（number；default = \'10\'）
- options.legend：保存图例名字的 table

本案例文件名为 PyTorch/Chapter06/pt06_vm04_line.py。

```
from visdom import Visdom
import numpy as np

vis = Visdom()

# line plots
Y = np.linspace(-5, 5, 100)
vis.line(
    Y=np.column_stack((Y * Y, np.sqrt(Y + 5))),
    X=np.column_stack((Y, Y)),
    opts=dict(markers=False),
)
```

运行脚本，显示效果如图 6.5 所示。

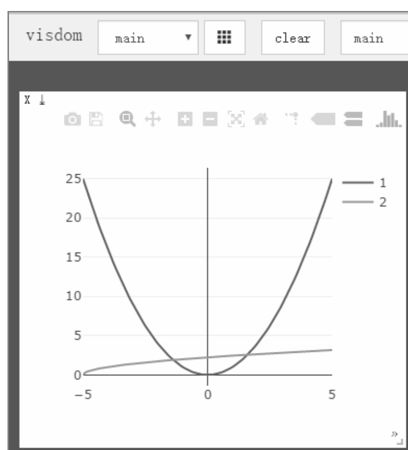


图 6.5　线图

6.4.6　vis.stem

这个函数用来画茎叶图。它需要一个形状为 N 或者 $N\times M$ 的 tensor $\boldsymbol{X}$ 来指定 M 时间序列中 N 个点的值。还需要一个可选择的 Y，形状为 N 或者 $N\times M$，用 Y 来指定时间戳，如果 Y 的形状是 N，那么默认 M 时间序列共享同一个时间戳。

支持以下特定选项。

- options.colormap：色图（string；default = \'Viridis\'）
- options.legend：保存图例名字的 table

本案例文件名为 PyTorch/Chapter06/pt06_vm05_stem.py。

```
from visdom import Visdom
import numpy as np
import math

vis = Visdom()

# stemplot
Y = np.linspace(0, 2 * math.pi, 70)
X = np.column_stack((np.sin(Y), np.cos(Y)))
vis.stem(
    X=X,
    Y=Y,
    opts=dict(legend=['Sine', 'Cosine'])
)
```

运行脚本，显示效果如图 6.6 所示。

6.4.7　vis.heatmap

这个函数用来画热力图。它输入一个形状为 $N\times M$ 的 tensor $\boldsymbol{X}$。$\boldsymbol{X}$ 指定了热力图中位置的值。

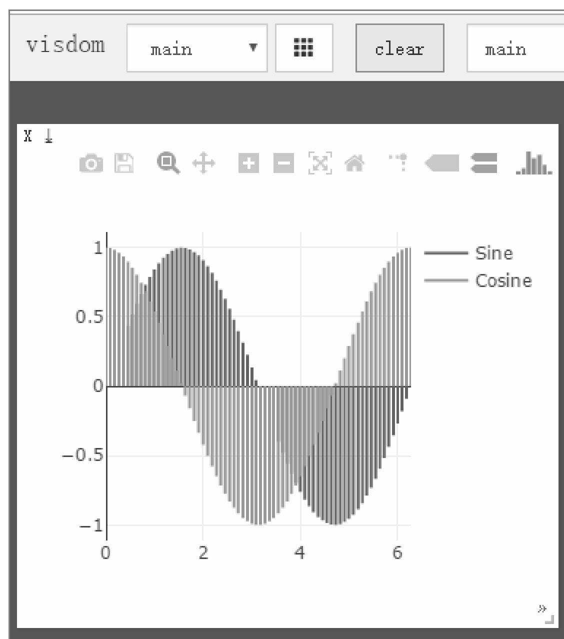


图 6.6　茎叶图

支持下列特定选项。

- options.colormap：色图（string；default = \’Viridis\’）
- options.xmin：小于这个值的会被剪切成这个值（number；default = X:min()）
- options.xmax：大于这个值的会被剪切成这个值（number；default = X:max()）
- options.columnnames：包含 X 轴标签的 table
- options.rownames：包含 Y 轴标签的 table

本案例文件名为 PyTorch/Chapter06/pt06_vm06_heatmap.py。

```
from visdom import Visdom
import numpy as np
import math

vis = Visdom()
```

```
    vis.heatmap(
        X=np.outer(np.arange(1, 6), np.arange(1, 11)),
        opts=dict(
            columnnames=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
'j'],
            rownames=['y1', 'y2', 'y3', 'y4', 'y5'],
            colormap='Electric',
        )
    )
```

运行脚本，显示效果如图 6.7 所示。

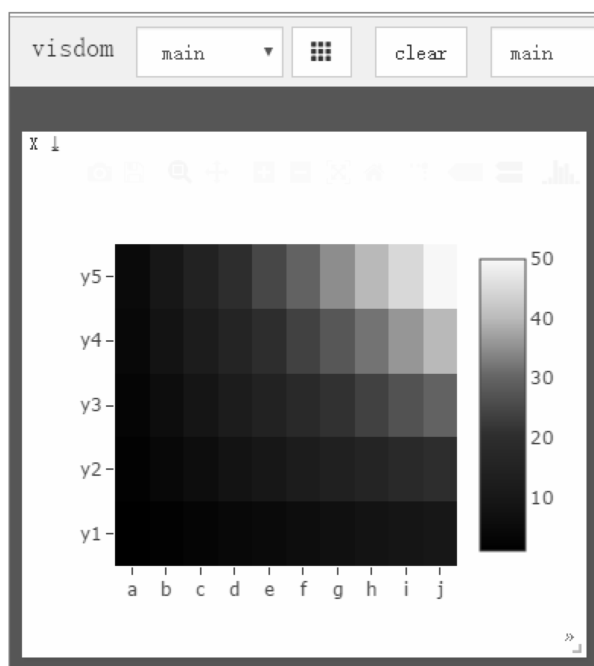


图 6.7　热力图

6.4.8　vis.bar

这个函数可以画普通的、堆起来的、分组的条形图。

输入参数如下。

- ***X***（tensor）：形状 *N* 或 *N*×*M*，指定每个条的高度。如果 ***X*** 有 *M* 列，那么每行的值可以看作一组或者把它们的值堆起来（取决与 options.stacked 是否为 True）。
- ***Y***（tensor，optional）：形状 *N*，指定对应的 X 轴的值。

支持以下特定选项。

- options.columnnames：包含 X 轴标签的表格
- options.stacked：在 ***X*** 中堆叠多个列
- options.legend：包含图例标签的表

本案例文件名为 PyTorch/Chapter06/pt06_vm07_bar.py。

```
from visdom import Visdom
import numpy as np
import math

vis = Visdom()

# 单个条形图
vis.bar(X=np.random.rand(20))

# 堆叠条形图
vis.bar(
    X=np.abs(np.random.rand(5, 3)),
    opts=dict(
        stacked=True,
        legend=['Sina', '163', 'AliBaBa'],
        rownames=['2013', '2014', '2015', '2016', '2017']
    )
)

# 分组条形图
vis.bar(
    X=np.random.rand(20, 3),
    opts=dict(
        stacked=False,
```

```
        legend=['A', 'B', 'C']
    )
)
```

运行脚本，显示效果如图 6.8 所示。

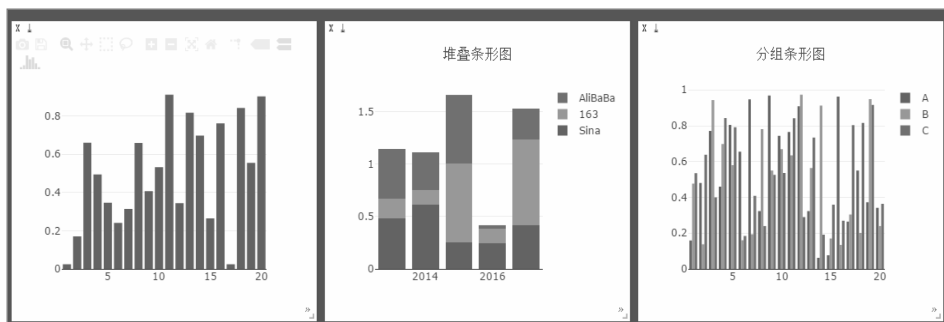


图 6.8　条形图

6.4.9　vis.histogram

这个函数用来画指定数据的直方图。它需要输入长度为 N 的 tensor ***X***。***X*** 保存了构建直方图的值。

支持下面特定选项。

- options.numbins：bins 的个数（number；default = 30）

本案例文件名为 PyTorch/Chapter06/pt06_vm06_heatmap.py。

```
from visdom import Visdom
import numpy as np

vis = Visdom()


vis.histogram(X=np.random.rand(10000), opts=dict(numbins=20))
```

运行脚本，显示效果如图 6.9 所示。

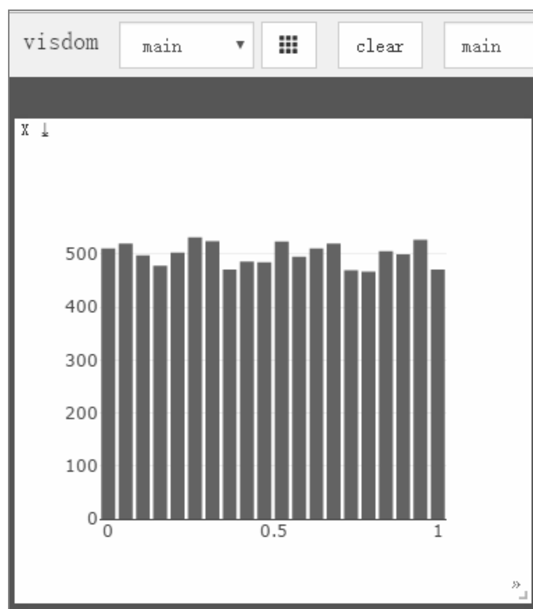


图 6.9　直方图

6.4.10　vis.boxplot

这个函数用来画箱型图，输入参数如下。

- ***X***（tensor）：形状 *N* 或 *N*×*M*，指定做第 *M* 个箱型图的 *N* 个值。

支持以下特定选项。

- options.legend：labels for each of the columns in ***X***

本案例文件名为 PyTorch/Chapter06/pt06_vm09_boxplot.py。

```
from visdom import Visdom
import numpy as np

vis = Visdom()

# boxplot
X = np.random.rand(100, 2)
X[:, 1] += 2
```

```
vis.boxplot(
    X=X,
    opts=dict(legend=['Men', 'Women'])
)
```

运行脚本，显示效果如图 6.10 所示。



图 6.10　箱型图

6.4.11　vis.surf

这个函数用来画表面图，输入参数如下。

- ***X***（tensor）：形状 $N \times M$，指定表面图上位置的值。

支持以下特定选项。

- options.colormap：colormap（string；default = \'Viridis\'）
- options.xmin：clip minimum value（number；default = X:min()）
- options.xmax：clip maximum value（number；default = X:max()）

本案例文件名为 PyTorch/Chapter06/pt06_vm10_surf.py。

```
from visdom import Visdom
import numpy as np

vis = Visdom()

x = np.tile(np.arange(1, 101), (100, 1))
y = x.transpose()
X = np.exp((((x - 50) ** 2) + ((y - 50) ** 2)) / -(20.0 ** 2))

vis.surf(X=X, opts=dict(colormap='Hot'))
```

运行脚本，显示效果如图 6.11 所示。

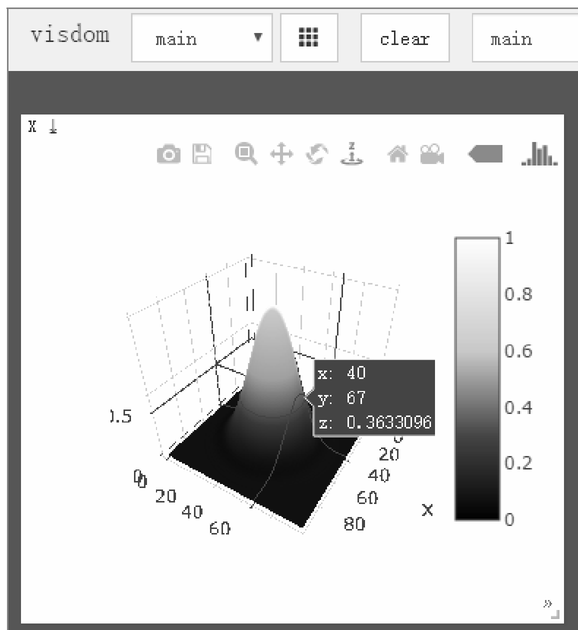


图 6.11　表面图

6.4.12　vis.contour

这个函数用来画轮廓图。输入参数如下。

- ***X***（tensor）：形状 $N \times M$，指定了轮廓图中的值。

支持以下特定选项。

- options.colormap：colormap（string；default = \'Viridis\'）
- options.xmin：clip minimum value（number；default = X:min()）
- options.xmax：clip maximum value（number；default = X:max()）

本案例文件名为 PyTorch/Chapter06/pt06_vm11_contour.py。

```
from visdom import Visdom
import numpy as np

vis = Visdom()

x = np.tile(np.arange(1, 101), (100, 1))
y = x.transpose()
X = np.exp((((x - 50) ** 2) + ((y - 50) ** 2)) / -(20.0 ** 2))

vis.contour(X=X, opts=dict(colormap='Viridis'))
```

运行脚本，显示效果如图 6.12 所示。

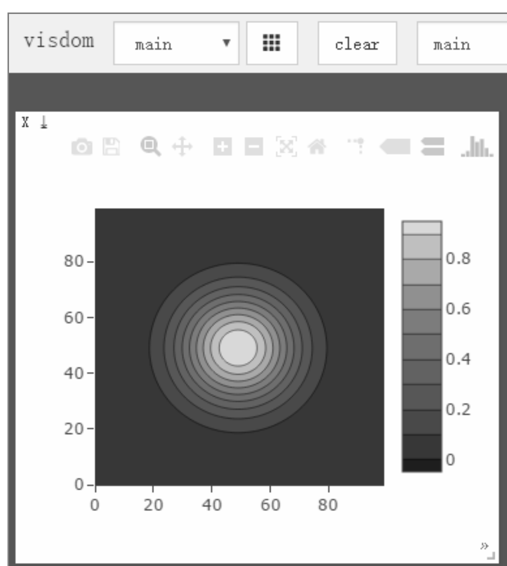


图 6.12　轮廓图

6.4.13 vis.mesh

此函数用来画出一个网格图。输入参数如下。

- ***X***（tensor）：shape（N×2 或 N×3）定义 N 个顶点。
- ***Y***（tensor，optional）：shape（M×2 或 M×3）定义多边形。

支持下列特定选项。

- options.color：color（string）
- options.opacity：多边形的不透明性（number between 0 and 1）

本案例文件名为 PyTorch/Chapter06/pt06_vm12_mesh.py。

```
from visdom import Visdom
import numpy as np

vis = Visdom()

# mesh plot
x = [0, 0, 1, 1, 0, 0, 1, 1]
y = [0, 1, 1, 0, 0, 1, 1, 0]
z = [0, 0, 0, 0, 1, 1, 1, 1]
X = np.c_[x, y, z]
i = [7, 0, 0, 0, 4, 4, 6, 6, 4, 0, 3, 2]
j = [3, 4, 1, 2, 5, 6, 5, 2, 0, 1, 6, 3]
k = [0, 7, 2, 3, 6, 7, 1, 1, 5, 5, 7, 6]
Y = np.c_[i, j, k]

vis.mesh(X=X, Y=Y, opts=dict(opacity=0.5))
```

运行脚本，显示效果如图 6.13 所示。

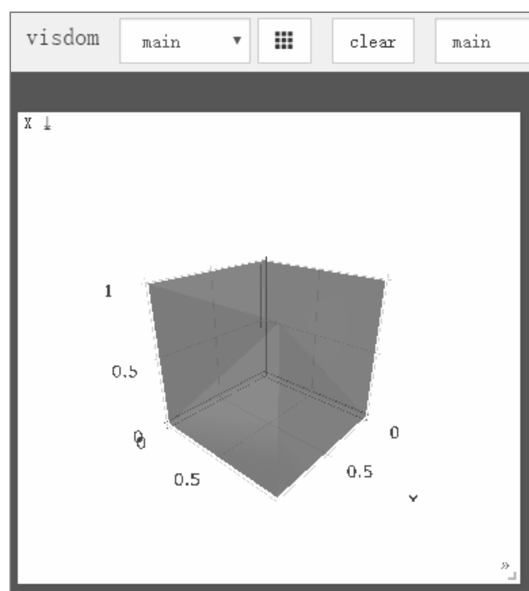


图 6.13　网格图

6.4.14　vis.svg

此函数绘制一个 SVG 对象。输入是一个 SVG 字符串或一个 SVG 文件的名称。

本案例文件名为 PyTorch/Chapter06/pt06_vm13_svg.py。

```
from visdom import Visdom
import numpy as np

vis = Visdom()

svgstr = """
<svg height="300" width="300">
  <ellipse cx="80" cy="80" rx="50" ry="30"
   style="fill:red;stroke:purple;stroke-width:2" />
  抱歉，你的浏览器不支持在线显示 SVG 对象.
</svg>
"""
```

```
vis.svg(
    svgstr=svgstr,
    opts=dict(title='SVG 图像')
)
```

运行脚本，显示效果如图 6.14 所示。

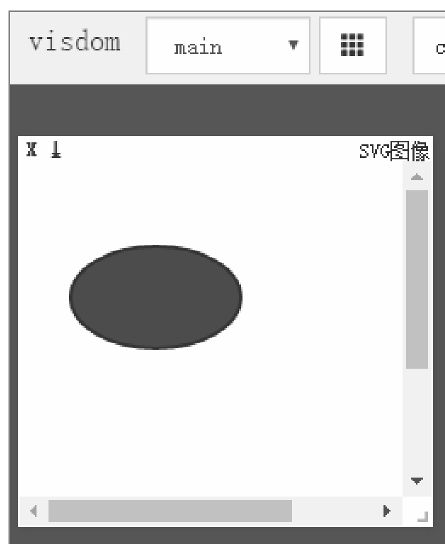


图 6.14　SVG 图像

2

第二部分

实战部分

第 7 章　卷积神经网络
第 8 章　循环神经网络简介
第 9 章　自编码模型
第 10 章　对抗生成网络
第 11 章　Seq2seq 自然语言处理
第 12 章　利用 PyTorch 实现量化交易

第 7 章 卷积神经网络

卷积神经网络是近年发展起来，并引起广泛重视的一种高效识别方法。

卷积神经网络是人工神经网络的一种，是深度学习的一个重要算法，它在很多应用上表现出卓越的效果。特别是在模式分类领域，由于该网络避免了对图像的复杂前期预处理，可以直接输入原始图像，因而得到了更为广泛的应用。

20 世纪 60 年代，Hubel 和 Wiesel 在研究猫脑皮层中用于局部敏感和方向选择的神经元时，发现其独特的网络结构可以有效地降低反馈神经网络的复杂性，继而提出了卷积神经网络（Convolutional Neural Networks，简称 CNN）。

由于 CNN 的特征检测层通过训练数据进行学习，因此可以隐式地从训练数据中进行学习。

CNN 主要用来识别位移、缩放及其他形式扭曲不变性的二维图形。

在图像处理过程中，由于图像像素可以看作是多维输入向量，同一特征映射面上的神经元权值相同，权值共享减少了权值的数量，降低了网络的复杂性，因此卷积神经网络以其局部权值共享的特殊结构在语音识别和图像处理方面有着独特的优越性。

同时卷积神经网络对输入图片的平移、比例缩放、倾斜或者其他形式

的变形具有高度不变性。

神经网络的基本组成包括输入层、隐藏层、输出层。卷积神经网络也不例外，包括输入层、隐藏层和输出层。但是卷积神经网络的隐藏层可分为卷积层和池化层。

卷积神经网络具有特殊深层的神经网络模型结构，它的特殊性体现在两个方面，一方面神经元之间的连接是非全连接的，另一方面同一层中某些神经元之间的连接的权重是共享的。它的非全连接和权值共享的网络结构使之更类似于生物神经网络，降低了网络模型的复杂度，减少了权值的数量。从卷积神经网络的构造上来看，卷积神经网络的基本结构包括两层，其一为特征提取层，每个神经元的输入与前一层的局部接受域相连，并提取该局部的特征。一旦该局部特征被提取后，它与其他特征间的位置关系也随之确定下来。其二为特征映射层，网络的每个计算层由多个特征映射组成，每个特征映射是一个平面，平面上所有神经元的权值相等。一般采用 Sigmoid 函数作为卷积网络的激活函数，这使得特征映射具有位移不变性。

图 7.1 是卷积神经网络的基本流程结构。

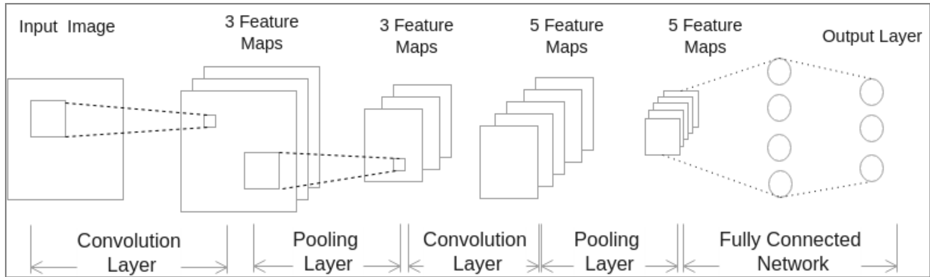


图 7.1　卷积神经网络结构

我们用文字简单描述一下卷积神经网络的基本流程。

一个卷积神经网络由若干卷积层、池化层、全连接层组成。

（1）输入图像到输入层。

（2）第一个卷积层的输入图像通过三个可训练的卷积核加偏置进行卷积操作，得到了三个特征图。

（3）在第一个卷积层之后，池化层对三个特征图做了下采样，得到了

三个更小的特征图。

（4）第二个卷积层，每个卷积核 Fitler 都把前面下采样之后的特征图卷积在一起，得到一个新的特征图，然后得到了 5 个特征图。

（5）第二个池化层，对 5 个特征图进行下采样，得到了 5 个更小的特征图。

（6）卷积神经网络的最后两层是全连接层。这些像素值连接成一个向量输入到传统的神经网络中，得到输出。

7.1　卷积层

卷积是分析数学中一种很重要的运算，这里面我们只介绍离散形式的卷积。在图像上，对图像用一个卷积核进行卷积运算，实际上是一个滤波的过程。卷积实际上是提供了一个权重模板，这个模板在图像上滑动，并将中心依次与图像中每一个像素对齐，然后对这个模板覆盖的所有像素进行加权，并将结果作为这个卷积核在图像上该点的响应。

如果输入的是一张图片，首先把输入图像分解成可以被卷积层处理的矩阵，把卷积核作用于输入的不同的区域，然后产生对应的特征图，如图 7.2 所示。

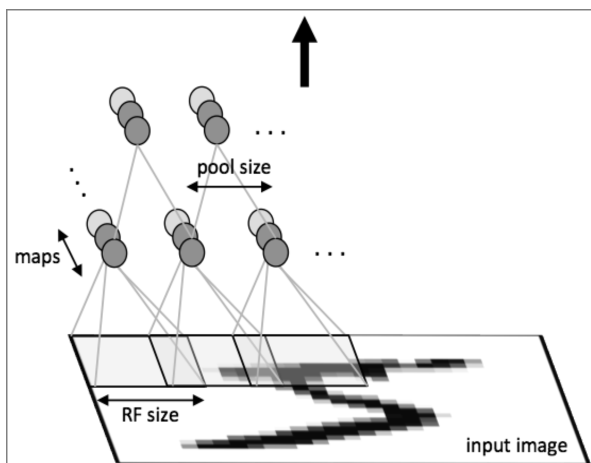


图 7.2　特征提取

下面我们来看一看具体怎么操作的？

关于卷积层我们先来看什么叫卷积操作：图 7.3 较大网格表示一幅图片，卷积核 Kj 的大小为 2×2。假设我们做步长为 1 的卷积操作，表示卷积核每次向右移动一个像素（当移动到边界时回到最左端并向下移动一个单位）。在卷积网络中，每个稀疏过滤器通过共享权值都会覆盖整个可视域，这些共享权值的单元构成一个特征映射卷积核。一方面，重复单元能够对特征进行识别，而不考虑它在可视域中的位置。另一方面，权值共享使得我们能更有效地进行特征抽取，因为它极大地减少了需要学习的自由变量的个数。通过控制模型的规模，卷积网络对视觉问题可以具有很好的泛化能力，每个单元内都有权重。

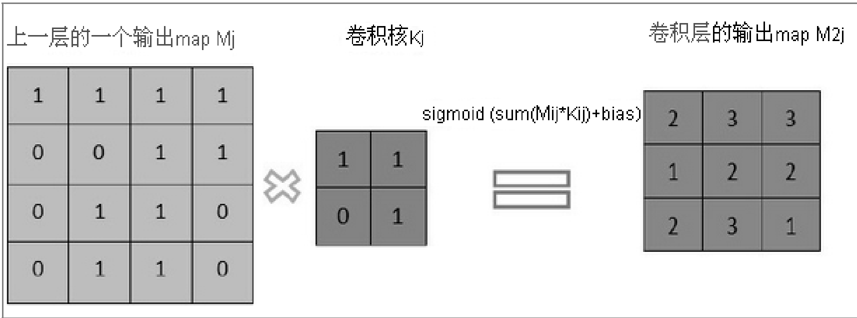


图 7.3　卷积运算

图 7.3 的卷积核内有 4 个权重。在卷积核移动的过程中，将图片上的像素和卷积核的对应权重相乘，最后将所有乘积相加得到一个个（4–2+1）×（4–2+1）=3×3 的特征图。假设上一层的图大小是 $n\times n$，卷积核的大小是 $k\times k$，则该层的图大小是（$n-k+1$）×（$n-k+1$）。

有时候我们会遇到每次移动步长较大，导致窗口的滑动出现不能刚好从头到尾的情况。于是就出现了 zero-padding 操作进行补零，保证窗口的滑动能刚好从头到尾。举个例子，（4–2+1）×（4–2+1）=3×3 刚好能够整除，所以窗口左侧贴着数据开始位置，滑到尾部刚好窗口右侧能够贴着数据尾部位置，因此是不需要补零的。如果滑动步长设为 3，第一次计算之后，窗口就无法滑动了，而尾部的数据，就没有计算，因此补零能够解决这个问题。我们可以发现，窗口滑动步长设定越短，两次滑动取得的数据，重叠

部分越多，但是窗口停留的次数也会越多，运算量大。窗口滑动步长设定越长，两次滑动取得的数据，重叠部分越少，窗口停留次数也越少，运算量小，但是从一定程度上说，数据信息不如上面丰富了。

7.2 池化层

下采样层也叫池化层，池化层最常用的方法包括最大池化和平均池化，主要是用来降低网络的复杂度。池化层的输入一般来源于上一个卷积层，主要作用是提供很强的鲁棒性并且减少参数的数量，防止过拟合现象的发生。如果取区域均值（Mean-pooling），往往能保留整体数据的特征，能突出背景的信息，而如果取区域最大值（Max-pooling），则能更好地保留纹理上的特征。例如 Max-pooling 是取一小块区域中的最大值，此时若此区域中的其他值略有变化，或者图像稍有平移，pooling 后的结果仍不变。

下采样层是对上一层特征图的一个采样处理，采用最大池化处理，采用 2×2 小区域的均值结果，如图 7.4 所示。

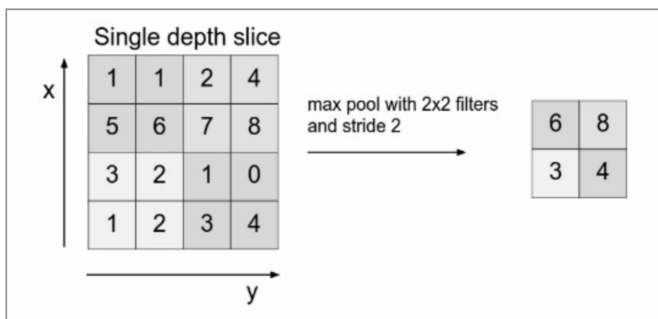


图 7.4　池化层运算

卷积神经网络采用反向传输调整权重，虽然看来基本思想跟 BP 神经网络算法一样，都是通过最小化残差来调整权重和偏置，但 CNN 的网络结构比较复杂，而且权重共享，使得计算残差变得很困难。卷积神经网络里面有时候会用到各种各样的归一化层，尤其是早期的研究，经常能见到它们的身影，不过近些年来的研究表明，似乎这个层级对最后结果的帮助非常小，有时候甚至干脆去掉不用了。

在卷积神经网络的最后，往往会出现一两层全连接层，全连接一般会把卷积输出的二维特征图转化成一维的一个向量，全连接层的每一个结点都与上一层的所有结点相连，用来把前边提取到的特征综合起来。由于其全相连的特性，一般全连接层的参数也是最多的。传统的网络我们的输出都是分类，也就是几个类别的概率，甚至就是一个类别，那么全连接层就是高度提纯的特征了，它方便最后的分类器或者回归使用。

7.3　经典的卷积神经网络

7.3.1　LeNet-5 神经网络结构

LeNet-5 是一种典型的用来识别数字的卷积网络。当年美国大多数银行就是用它来识别支票上面的手写数字的，如图 7.5 所示。

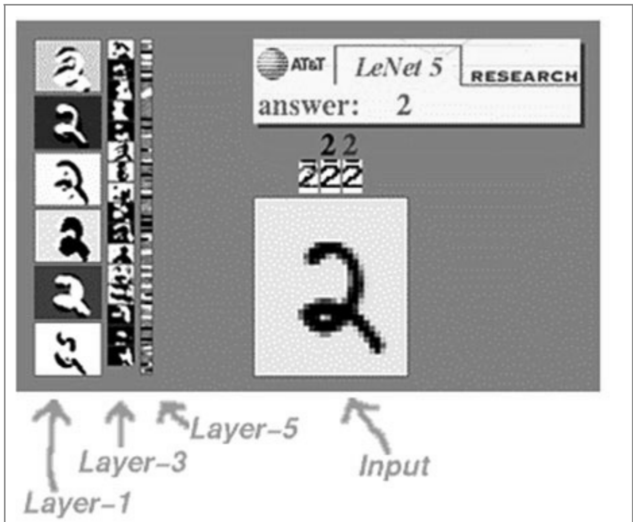


图 7.5　LeNet-5 神经网络识别手写数字

LeNet-5 共有 7 层，不包含输入，每层都包含可训练参数（连接权重），如图 7.6 所示。输入图像大小为 32×32。

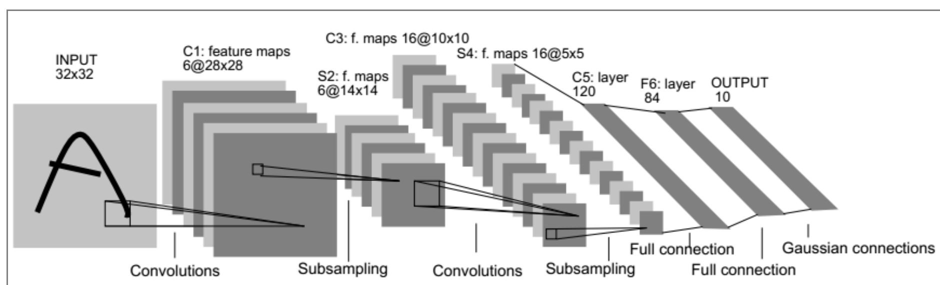


图 7.6 LeNet-5 神经网络结构

输入层 Input 到 C1 之间：Input 为 32×32，通过 5×5 的卷积运算得到，由 6 个特征图构成。特征图每层为(32–5+1)×(32–5+1)=28×28。由于特征图相同层共享权值，而不同层之间权值不同，因此，共有 6×(5×5+1)=156 个参数。

C1 到 S2 之间：S2 层是一个下采样层，有 6 个 14×14 的特征图，S2 中的每个点对应 C1 中 2×2 的区域，再乘以一个系数，再加上一个偏差，结果通过 Sigmoid 函数计算。因此这层共有 6×(1+1)=12 个参数。

S2 到 C3 之间：有 16 种不同的卷积核，所以就存在 16 个特征图。C3 总共有 16 层，这层共有 1516=5×5×(6×3+9×4+1×6)+16，其中最后一项 16 代表 16 个偏差。

C3 到 S4 之间：S4 层是一个下采样层，由 16 个 5×5 大小的特征图构成。

C5 是一个卷积层，共有 120 个特征图，每个特征图的尺寸是 1×1，C5 的每个单元与 S4 的 16 个 Feature Map 相连。

F6 有 84 个单元，与 C5 全连接，共有(120+1)×84=10164。F6 层的计算都是通过输入向量和权值向量相乘，并加偏置，最后通过一个 Sigmoid 函数计算。

最后的输出层由欧几里得 RBF 单元组成，每个单元代表一个类别，输入向量与权值向量相差越大，RBF 输出越大。F6 层为 84 个元素输出。

7.3.2　ImageNet-2010 网络结构

与计算机视觉顶会 CVPR 2017 同期举行的 Workshop——“超越 ILSVRC”（Beyond ImageNet Large Scale Visual Recognition Challenge），宣布计算机视觉乃至整个人工智能发展史上的里程碑——IamgeNet 大规模视觉识别挑战赛将于 2017 年正式结束，此后将专注于目前尚未解决的问题及以后的发展方向。

截至 2016 年年末，ImageNet 中含有超过 1500 万个由人手工注释的图片网址，也就是带标签的图片，标签说明了图片中的内容，超过 2.2 万个类别。其中，至少有 100 万张里面提供了边框（Bounding Box）。ImageNet 项目仍在进行中，目前有来自 21841 个不同类别的 14197122 张图像。自 2010 年以来，ImageNet 举办了视觉识别领域的年度竞赛赛事，为参赛者提供来自 1000 个不同类别的 120 万张图像。因此，每个网络架构的准确率都建立在这 1200 万张图像之上，如图 7.7 和图 7.8 所示。

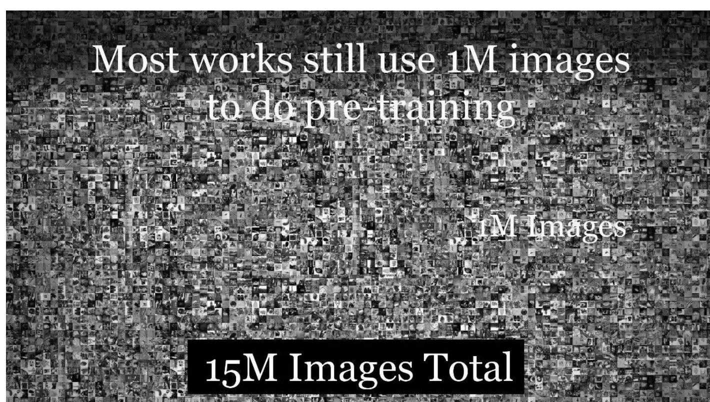


图 7.7　ImageNet 图像

ImageNet LSVRC 是一个图片分类的比赛，其训练集包括 127 万张图片，验证集有 5 万张图片，测试集有 15 万张图片。本文截取 2010 年 Alex Krizhevsky 的 CNN 结构进行说明，该结构在 2010 年取得冠军，top-5 错误率为 15.3%。值得一提的是，在 2017 年的 ImageNet LSVRC 比赛中，取得冠军的 GoogNet 已经达到了 top-5 错误率 6.67%。可见，深度学习的提升空间还很巨大。

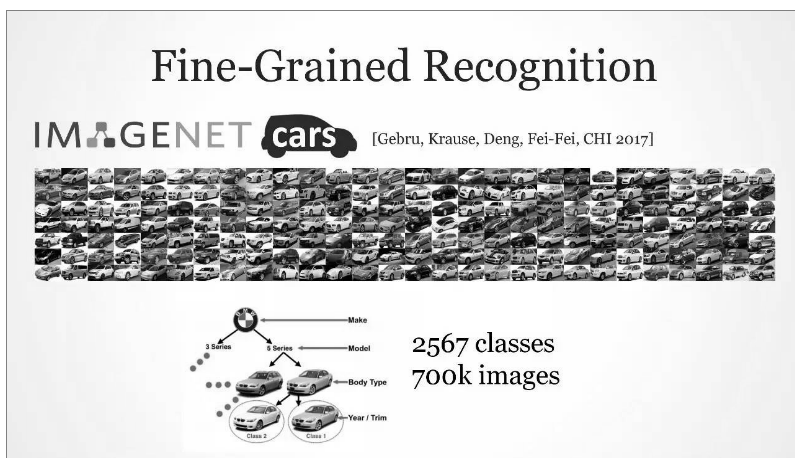


图 7.8　ImageNet 分类图像

图 7.9 即为 Alex 的 CNN 结构图。需要注意的是，该模型采用了 2-GPU 并行结构，即第 1、2、4、5 卷积层都是将模型参数分为两部分进行训练的。在这里，更进一步，并行结构分为数据并行与模型并行。数据并行是指在不同的 GPU 上，模型结构相同，但将训练数据进行切分，分别训练得到不同的模型，然后再将模型进行融合。而模型并行则是，将若干层的模型参数进行切分，不同的 GPU 上使用相同的数据进行训练，得到的结果直接连接作为下一层的输入。

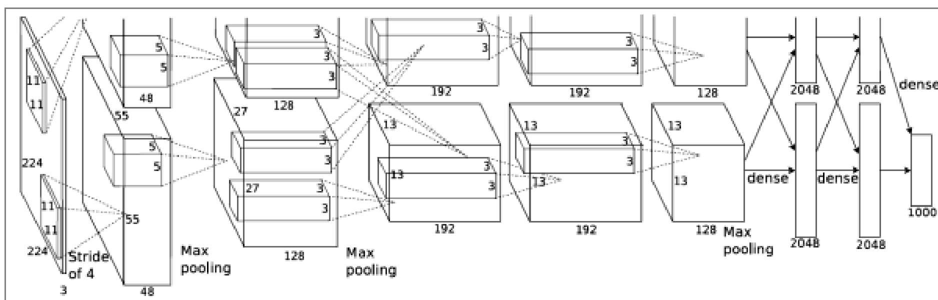


图 7.9　Alex 的 CNN 结构图

图 7.9 模型的基本参数如下。

输入：224×224 大小的图片，3 通道。

第一层卷积：5×5 大小的卷积核 96 个，每个 GPU 上 48 个。

第一层 Max-pooling：2×2 的核。

第二层卷积：3×3 卷积核 256 个，每个 GPU 上 128 个。

第二层 Max-pooling：2×2 的核。

第三层卷积：与上一层是全连接，3×3 的卷积核 384 个。分到两个 GPU 上各 192 个。

第四层卷积：3×3 的卷积核 384 个，两个 GPU 各 192 个。该层与上一层连接没有经过池化层。

第五层卷积：3×3 的卷积核 256 个，两个 GPU 上各 128 个。

第五层 Max-pooling：2×2 的核。

第一层全连接：4096 维，将第五层 Max-pooling 的输出连接成为一个一维向量，作为该层的输入。

第二层全连接：4096 维。

Softmax 层：输出为 1000，输出的每一维都是图片属于该类别的概率。

AlexNet 相比传统的 CNN（比如 LeNet）有哪些重要改动呢？如图 7.10、图 7.11 和图 7.12 所示。

1. 水平翻转

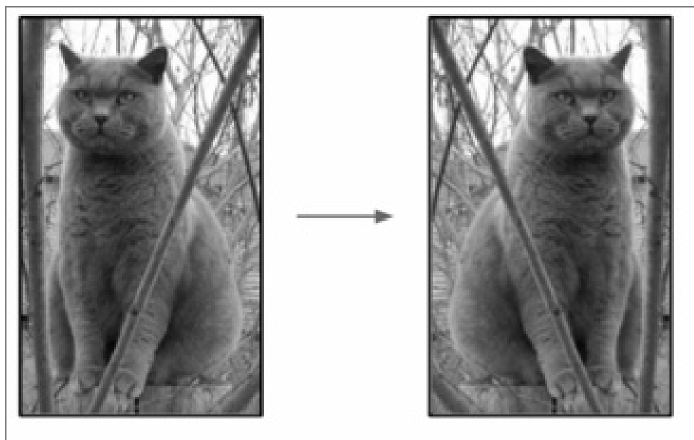


图 7.10　猫水平翻转图片

2. 随机裁剪、平移变换

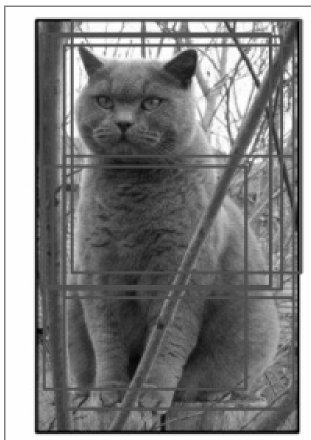


图 7.11　随机裁剪、平移变换后的图像

3. 颜色、光照变换

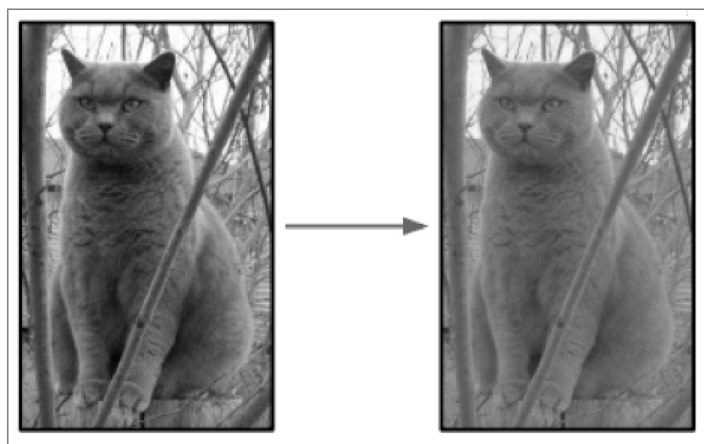


图 7.12　颜色、光照变换后的图像

AlexNet 由 Alex Krizhevsky 提出，使用 ReLU 处理非线性的部分，而非传统神经网络早期的标准是Tanh 或 Sigmoid 函数。ReLU 的公式为：$f(x) = \max(0, x)$。

ReLU 与 Sigmoid 相比，其优势在于训练速度更快，因为 Sigmoid 的导数在饱和区变得很小，导致权重几乎没有得到更新（如图 7.13 所示）。这

种情况就是梯度消失问题。

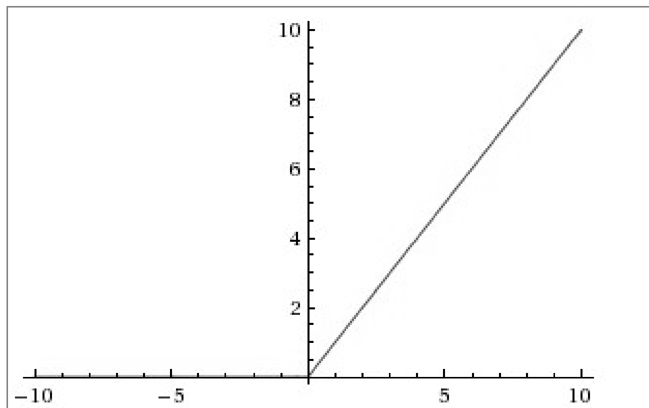


图 7.13　ReLU 函数

Dropout 应该算是 AlexNet 中一个很大的创新方法，Dropout 方法和数据增强一样，都是防止过拟合的。同时，用 ReLU 代替了传统的 Tanh 或者 Logistic。在每个全连接层后面使用一个 Dropout 层，从而减少过拟合。Dropout 层设置的概率为 p，表示每个神经元连接到后层神经元的概率为 1-p。该架构以概率 p 随机关闭激活函数，如图 7.14 和图 7.15 所示。

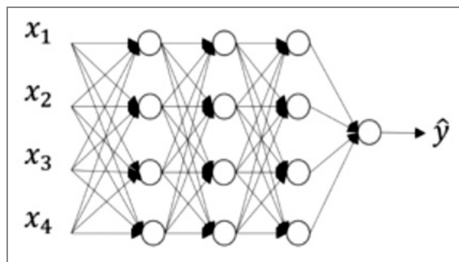


图 7.14　未经过舍弃神经元的模型

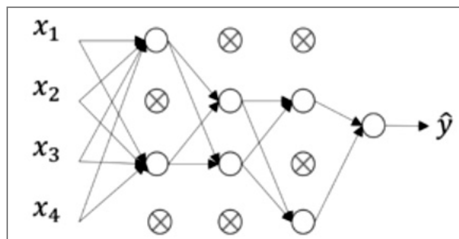


图 7.15　经过舍弃神经元的模型

AlexNet 中用到了一些非常大的卷积核，比如 11×11、5×5 卷积核，之前人们的观念是，卷积核越大，Receptive Field（感受野）越大，看到的图片信息越多，因此获得的特征越好。虽说如此，但是大的卷积核会导致计算量的暴增，不利于模型深度的增加，计算性能也会降低。于是在 VGG（最早使用）、Inception 网络中，利用 2 个 3×3 卷积核的组合比 1 个 5×5 卷积核的效果更佳，同时参数量（3×3×2+1 vs 5×5×1+1）被降低，因此后来 3×3 卷积核被广泛应用在各种模型中。

7.3.3　VGGNet 网络结构

VGGNet 由牛津大学的视觉几何组（Visual Geometry Group）提出，是 ILSVRC 2014 中定位任务第一名和分类任务第二名。其突出贡献在于证明使用很小的卷积（3×3），增加网络深度可以有效提升模型的效果，而且 VGGNet 对其他数据集具有很好的泛化能力。图 7.16 显示的是 VGGNet 各级别的网络结构图。

VGGNet 拥有 5 段卷积，每一段卷积 2~3 个卷积层，同时每段尾部会连接一个最大池化层来缩小图片尺寸。VGGNet 一个重要的改变就是使用 3×3 的卷积核，用多个卷积核串联来减少总的参数，同时增加非线性变换的能力。

VGGNet-16 网络结构有六段，前五段是卷积网络，最后一段是全连接网络。

第一段卷积网络，两个卷积层（conv_op），一个最大池化层（mpool_op）。卷积核大小 3×3，卷积核数量（输出通道数）64，步长 1×1，全像素扫描。第一卷积层输入 input_op 尺寸 224×224×3，输出尺寸 224×224×64。第二卷积层输入输出尺寸 224×224×64。最大池化层 2×2，输出尺寸 112×112×64。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 7.16　VGGNet-16 结构图

第二段卷积网络，2 个卷积层，1 个最大池化层。卷积输出通道数 128。输出尺寸 56×56×128。

第三段卷积网络，3 个卷积层，1 个最大池化层。卷积输出通道数 256。输出尺寸 28×28×256。

第四段卷积网络，3 个卷积层，1 个最大池化层。卷积输出通道数 512。输出尺寸 14×14×512。

第五段卷积网络，3 个卷积层，1 个最大池化层。卷积输出通道数 512。输出尺寸 7×7×512。

输出结果是每个样本扁平化长度为 7×7×512=25088 的一维向量。

然后连接 4096 隐含点全连接层，激活函数 ReLU。所有隐藏层都使用 ReLU 函数。

最后连接 1000 隐含点全连接层，Softmax 分类输出概率。

在 AlexNet 基础上将单层网络替换为堆叠的 3×3 的卷积层和 2×2 的最大池化层，减少卷积层参数，同时加深网络结构，提高性能。

采用 Pre-trained 方法，利用浅层网络（A）训练参数，初始化深层网络参数（D，E），加速收敛。

采用 Multi-Scale 方法进行数据增强、训练、测试，提高准确率。

去掉 LRN，减少内存的消耗和计算时间。

与 ILSVRC-2012 和 ILSVRC-2013 最好的结果相比，VGGNet 优势很大。与 GoogLeNet 对比，虽然 7 个网络集成效果不如 GoogLeNet，但是单一网络测试误差好一些，而且只用 2 个网络集成效果就与 GoogLeNet 的 7 个网络集成效果差不多。效果表现如图 7.17 所示。

Performance on the Places205 dataset

Model	top-1 val/test	top-5 val/test
Places205-VGGNet-11	58.6/59.0	87.6/87.6
Places205-VGGNet-13	60.2/60.1	88.1/88.5
Places205-VGGNet-16	60.6/60.3	88.5/88.8
Places205-VGGNet-19	61.3/61.2	88.8/89.3

图 7.17　不同层数 VGGNet 的效果表现

7.3.4　GoodLeNet 网络结构

GoogLeNet，2014 年 ILSVRC 挑战赛冠军，将 Top5 的错误率降低到 6.67%。

GoogLeNet 是一个 22 层的深度网络。特点有如下。

1．网络包含 22 个带参数的层。一般来说，提升网络性能最直接的办法就是增加网络深度和宽度，这也就意味着巨量的参数。但是，巨量参数容易产生过拟合，会大大增加计算量。

2．网络的感受野大小是 224×224，采用 RGB 彩色通道，且减去均值。

3．网络中间的层次生成的特征非常有区分性，可以给这些层增加一些辅助分类器。采用均值池化层，滤波器大小为 5×5，步长为 3，（4a）的输出为 4×4×512，（4d）的输出为 4×4×528。1×1 的卷积由用于降维的 128 个滤波器和修正线性激活。全连接层有 1024 个单元和修正线性激活，Dropout 层的 Dropped 的输出比例为 70%。线性层将 Softmax 损失作为分类器。

在 3×3 和 5×5 的卷积前用一个 1×1 的卷积来减少计算，修正线性激活。显著增加每一步的单元数目，计算复杂性可受控制。

采用了 Inception 模块的网络要比没有采用 Inception 模块的网络快 2～3 倍。

GoogLeNet（22 层）结构如图 7.18 所示。

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

图 7.18　GoogLeNet（22 层）结构

ILSVRC 2014 的分类任务有 1000 个子类，120 万张训练图像，5 万张验证图像，10 万张测试图像，每张图像中有一个 Ground Truth，性能测量是基于得分最高的分类器预测的，常用指标为 top-1 准确率和 top-5 错误率。

GoogLeNet 最终的 top-5 错误率在验证集和测试集上都是 6.67%，获得了第一。与其他方法相比的提高如图 7.19 所示。

Team	Year	Place	Error (top-5)	Uses external data
SuperVision	2012	1st	16.4%	no
SuperVision	2012	1st	15.3%	Imagenet 22k
Clarifai	2013	1st	11.7%	no
Clarifai	2013	1st	11.2%	Imagenet 22k
MSRA	2014	3rd	7.35%	no
VGG	2014	2nd	7.32%	no
GoogLeNet	2014	1st	6.67%	no

图 7.19　各种模型的测试结果

7.3.5　ResNet 网络结构

ResNet——MSRA 何凯明团队的 Residual Networks，在 2015 年 ImageNet 上大放异彩，在 ImageNet 的 Classification、Detection、Localization 以及 COCO 的 Detection 和 Segmentation 上均斩获了第一名的成绩，而且 *Deep Residual Learning for Image Recognition* 也获得了 CVPR2016 的最佳论文。

深度学习网络常见问题。

① 深度学习网络的深度越深，常规的网络的堆叠（Plain Network）效果越不好。

② 网络越深，梯度消失的现象就越来越明显，网络的训练效果也不会很好。

③ 浅层的网络（Shallower Network）无法明显提升网络的识别效果，如图 7.20 所示。

ResNet 引入了残差网络结构（Residual Network），通过残差网络，可以把网络层做得很深，据说现在达到了 1000 多层，最终的网络分类的效果也非常好，残差网络的基本结构如图 7.21 所示。

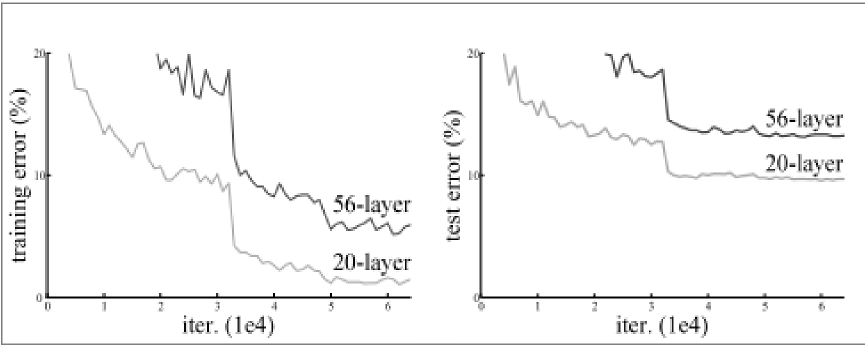


图 7.20　ResNet 模型表现

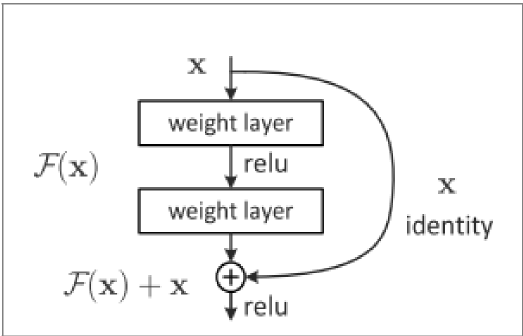


图 7.21　残差网络的基本结构

ResNet 用神经网络分解来降低拟合函数的复杂度，而损失函数是拟合函数的函数，所以降低拟合函数的复杂度也等同于降低了损失函数的复杂度。假设我们需要拟合的复杂函数为 H(x)，现在我们把 H(x)分解为两个更简单的函数 f(x)和 g(x)，即令 H(x) = f(x) + g(x)。这相当于对于相同数量的层减少了参数量，因此可以拓展成更深的模型。于是作者提出了 50、101、152 层的 ResNet，而且不仅没有出现退化问题，错误率也大大降低，同时计算复杂度也保持在很低的程度。

图 7.22 是 ResNet 网络结构，相当于在普通网络上面插入跳跃结构，对于跳跃结构，当输入与输出的维度一样时，不需要做其他处理，两者相加即可，但当两者维度不同时，输入要进行变换以后去匹配输出的维度，主要经过两种方式，用 Zero-padding 增加维度以及用 1x1 卷积来增加维度。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

图 7.22　ResNet 网络结构

残差网络结构简单，解决了极深度条件下深度卷积神经网络性能退化的问题，分类性能表现出色。残差网络的广泛使用已推进计算机视觉各任务的性能达到了新的高度。

残差网络表现出的良好图像分类性能，同样也可以进一步推广到人脸识别领域上来。使用残差网络，可以大幅提升人脸分类性能。在人脸识别准确率每 3 个月提升一个数量级的今天，残差网络及未来更高性能的网络结构必定会将这个周期进一步缩短。

AlexNet、VGG、GoogLeNet、ResNet 对比，如图 7.23 所示。

模型名	AlexNet	VGG	GoogLeNet	ResNet
初入江湖	2012	2014	2014	2015
层数	8	19	22	152
Top-5错误	16.4%	7.3%	6.7%	3.57%
Data Augmentation	+	+	+	+
Inception(NIN)	–	–	+	–
卷积层数	5	16	21	151
卷积核大小	11,5,3	3	7,1,3,5	7,1,3,5
全连接层数	3	3	1	1
全连接层大小	4096,4096,1000	4096,4096,1000	1000	1000
Dropout	+	+	+	+
Local Response Normalization	+	–	+	–
Batch Normalization	–	–	–	+

图 7.23　AlexNet、VGG、GoogLeNet、ResNet 对比

最适合 CNN 的莫过于分类任务，如语义分析、垃圾邮件检测和话题分类。卷积运算和池化会丢失局部区域某些单词的顺序信息，因此纯 CNN 的结构框架不太适用于 PoS Tagging 和 Entity Extraction 等顺序标签任务。LeNet 主要是用于识别 10 个手写数字，当然，只要稍加改造也能用在 ImageNet 数据集上，但效果较差。

7.4　卷积神经网络案例

通过对上述经典卷积神经网络模型的学习，我们已经了解了经典卷积神经网络的结构，下面我们通过案例来实现经典卷积神经网络。

本例文件名为 PyTorch/Chapter07/pt01_convolutional_neural_network.py。

```
import torch
#导入 torch 包
import torch.nn as nn
#导入 torch 的 nn 包，里面包含神经网络所需要的函数
from torch.autograd import Variable
#导入自动求导机制
import torch.utils.data as Data
#导入 data 数据集
import torchvision
#导入 torchvision 数据集包，里面包含图像翻转等功能
import matplotlib.pyplot as plt
#导入画图包
#定义超参数
EPOCH = 3
BATCH_SIZE = 50
LR = 0.001
DOWNLOAD_MNIST = True

# 获取训练集 dataset
train_data = torchvision.datasets.MNIST(
            root='./mnist/',
            train=True,
            transform=torchvision.transforms.ToTensor(),
```

```
            download=DOWNLOAD_MNIST,
            )
```

1．数据预处理

参数说明如下。

- root：数据集，存在于根目录 processed/training.pt 和 processed/test.pt 中。
- train：True = 训练集，False = 测试集
- download：如果为 True，请从 Internet 下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。
- transform：将原数据规范化到（0，1）区间，数据变换（0，255）≥（0，1）。
- 由于加载的数据集为 array 格式，需要转换成 torch 能识别的 tensor，利用 torchvision.datasets 加载 DataLoader 中的 dataset 格式的数据，同时在获取的时候直接转换成训练所需的数据格式。

```
# 打印 MNIST 数据集的训练集及测试集的尺寸
print(train_data.train_data.size())
#输出数据集的图片数量，以及尺寸大小
torch.Size([60000, 28, 28])
#手写数字数据集的大小为 6000 张，大小为 28×28
print(train_data.train_labels.size())
#输出手写数字对应的标签
# torch.Size([60000])
```

输出的结果为 60000，手写数字数据集每张图片分别对应相应的标签，手写数字图片数量为 6000 张，对应标签为 6000 个。

```
for i in range(1,4):
    plt.imshow(train_data.train_data[i].numpy(), cmap='gray')
    plt.title('%i' % train_data.train_labels[i])
plt.show()
```

输出前 3 张照片，我们需要把照片的像素 torch 的 tensor 数据转换成 Numpy 的形式，才能输出照片。同时，输出照片对应的标签，用来实现图

片上的数字与真实数字做对比。我们来看输出结果（这里只输出一张），如图 7.24 所示。

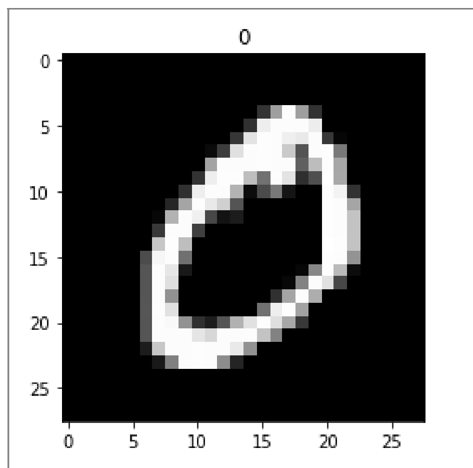


图 7.24　0 的手写照片

```
    #输出第一张照片是 0 的手写照片，对应的标签也是 0
    train_loader = Data.DataLoader(dataset=train_data,
batch_size=BATCH_SIZE,
                              shuffle=True)
    #我们已经下载好了 mnist 数据集，现在我们用 DataLoader 加载数据集
    torch.utils.data.DataLoader(dataset,batch_size=1,
shuffle=False, sampler=None, num_workers=0, collate_fn=<function
default_collate>, pin_memory=False, drop_last=False)
```

参数说明如下。

- dataset：加载数据的数据集。
- batch_size：加载批训练的数据个数。
- shuffle：True 在每个 epoch 重新排列的数据。
- sampler：从数据集中提取样本。
- batch_sampler：一次返回一批索引。
- num_workers：用于数据加载的子进程数。0 表示数据将在主进程中加载。
- collate_fn：合并样本列表以形成小批量。

- pin_memory：如果为 True，数据加载器在返回前将张量复制到 CUDA 固定内存中。
- drop_last：如果数据集大小不能被 batch_size 整除，设置为 True，可删除最后一个不完整的批处理。如果设为 False，并且数据集的大小不能被 batch_size 整除，则最后一个 batch 将更小。

```
    # 获取测试集 dataset，通过 torchvision.datasets 获取的 dataset 格式可
直接可置于 DataLoader
    test_data = torchvision.datasets.MNIST(root='./mnist/',
train=False)
    #加载测试集
    test_x = Variable(torch.unsqueeze(test_data.test_data, dim=1),
                 volatile=True).type(torch.FloatTensor)
    test_y = test_data.test_labels
```

2. 定义网络结构

① class CNN 需要继承 Module。

② 需要调用父类的构造方法：super(CNN, self).__init__()。

③ 在 PyTorch 中激活函数 ReLU 也算是一层 layer。

④ 需要实现 forward()方法，用于网络的前向传播，而反向传播只需要调用 Variable.backward()即可。只需定义 forward 函数，并 backward 自动使用函数 Autograd，可以在 forward 功能中使用任何 Tensor 操作。

输入层≥二维特征卷积≥Sigmoid 激励≥均值池化≥全连接网络≥Softmax 输出

```
    class CNN(nn.Module):
        def __init__(self):
            super(CNN, self).__init__()
            self.conv1 = nn.Sequential(
                         nn.Conv2d(in_channels=1, out_channels=16,
kernel_size=5,
                                   stride=1, padding=2),
                         nn.ReLU(),
                         nn.MaxPool2d(kernel_size=2) # (16,14,14)
                         )
```

```
        self.conv2 = nn.Sequential( # (16,14,14)
                     nn.Conv2d(16, 32, 5, 1, 2), # (32,14,14)
                     nn.ReLU(),
                     nn.MaxPool2d(2) # (32,7,7)
                     )
        self.out = nn.Linear(32*7*7, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1) # 将(batch, 32,7,7)展平为(batch,
32*7*7)
        output = self.out(x)
        return output

cnn = CNN()
#Conv2d(in_channels, out_channels, kernel_size, stride=1,
padding=0, dilation=1, groups=1, bias=True)
```

3．二维卷积层

参数 kernel_size、stride、padding、dilation 也可以是一个 int 的数据，此时卷积 height 和 width 值相同；也可以是一个 tuple 数组，tuple 的第一维度表示 height 的数值，tuple 的第二维度表示 width 的数值。

参数说明如下。

in_channels（int）：输入信号的通道。

out_channels（int）：卷积产生的通道。

kerner_size（int or tuple）：卷积核的尺寸。

stride（int or tuple,optional）：卷积步长。

padding（int or tuple,optional）：是否对输入数据填充 0。Padding 可以将输入数据的区域改造成是卷积核大小的整数倍，这样对不满足卷积核大小的部分数据就不会忽略了。通过 padding 参数指定填充区域的高度和宽度。

dilation（int or tuple，`optional``）：卷积核之间的空格。

groups（int，optional）：将输入数据分成组，in_channels 应该被组数整除。group=1，输出是所有的输入的卷积；group=2，此时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

bias（bool，optional）：如果 bias=True，添加偏置。

Pooling 函数主要是用于图像处理的卷积神经网络中，但随着深层神经网络的发展，Pooling 函数相关技术在其他领域，其他结构的神经网络中也越来越受关注。

卷积神经网络中的卷积层是对图像的一个邻域进行卷积得到图像的邻域特征，亚采样层就是使用 Pooling 函数技术将小邻域内的特征点整合得到新的特征。Pooling 函数确实起到了整合特征的作用。

池化操作是利用一个矩阵窗口在张量上进行扫描，在每个矩阵中通过取最大值或者平均值等来减少元素的个数，最大值和平均值的方法可以使得特征提取拥有“平移不变性”，也就是在图像有了几个像素的位移情况下，依然可以获得稳定的特征组合，因为平移不变性对于识别十分重要。

Pooling 函数的结果是使得特征减少，参数减少，但 pooling 的目的并不仅在于此。Pooling 函数的目的是为了保持某种不变性（旋转、平移、伸缩等），常用的有 Mean-pooling 函数，Max-pooling 函数和 Stochastic-pooling 函数三种。

MaxPool2d(kernel_size,stride,padding,ceil_mode,count_include_pad)，对由几个输入平面组成的输入信号进行二维最大池化。

参数说明如下。

kernel_size：窗口的大小。

stride：窗口的步长。默认值为 kernel_size。

Padding：是否对输入数据填充 0。padding 可以将输入数据的区域改造成是卷积核大小的整数倍，这样对不满足卷积核大小的部分数据就不会忽

略了。通过 padding 参数指定填充区域的高度和宽度。

ceil_mode：当为 True 时，将使用 ceil 代替 floor 来计算输出形状。

count_include_pad：当为 True 时，将包括平均计算中的零填充。默认值为 True。

输出卷积神经网络的搭建通过定义一个 CNN 类来实现，卷积层 conv1、conv2 及 out 层以类属性的形式定义，各层之间的衔接信息在 forward 中定义，定义的时候要留意各层的神经元数量。通过输出卷积神经网络的参数可以看出 2 维卷积的滤波器的大小为 5×5，卷积移动步长为 1 个单元，并且对数据填充，这样对不满足卷积核大小的部分数据就不会忽略了。通过 padding 参数指定填充区域的高度和宽度。池化函数的最大池化的窗口大小为 2×2，最大池化的窗口移动的步长为 2×2，dilation 控制移动窗口的步长大小为 1。

```
    print(cnn)
    '''''
    CNN (
      (conv1): Sequential (
        (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2))
        (1): ReLU ()
        (2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
      )
      (conv2): Sequential (
        (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2))
        (1): ReLU ()
        (2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
      )
      (out): Linear (1568 -> 10)
    )
    '''



    params = list(net.parameters())
    print(len(params))
```

```
print(params[0].size())
#打印网络结构的参数
optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)
```

torch.optim 提供了很多种更新方法，如 SGD、Nesterov－SGD、Adam、RMSProp 等，使用起来很简单。

Adam 算法可以看作是修正后的算法，Adam 优化算法是随机梯度下降算法的扩展式，近来其广泛用于深度学习应用中，尤其是计算机视觉和自然语言处理等任务。Adam 是一种可以替代传统随机梯度下降过程的一阶优化算法，它能基于训练数据迭代更新神经网络权重。

Adam 最开始是由 OpenAI 的 Diederik Kingma 和多伦多大学的 Jimmy Ba 在 2015 年 ICLR 论文 *Adam：A Method for Stochastic Optimization* 中提出。Adam 通常被认为对超参数的选择具有鲁棒性，Adam 通过计算梯度的一阶矩估计和二阶矩估计而为不同的参数设计独立的自适应性学习速率。学习速率建议为 0.001。Adam 在深度学习领域内是十分流行的算法，因为它能很快地实现优良的结果。经验性结果证明 Adam 算法在实践中性能优异，相对于其他种类的随机优化算法具有很大的优势。

SGD 指 Stochastic Gradient Descent，即随机梯度下降。对于训练数据集，我们首先将其分成 n 个 batch，每个 batch 包含 m 个样本。我们每次更新都利用一个 batch 的数据，而非整个训练集。当训练数据太多时，利用整个数据集更新往往时间上不显示。batch 的方法可以减少机器的压力，并且可以更快地收敛。当训练集有很多冗余时（类似的样本出现多次），batch 方法收敛更快。以一个极端情况为例，若训练集前一半和后一半梯度相同，如果前一半作为一个 batch，后一半作为另一个 batch，那么在一次遍历训练集时，batch 的方法向最优解前进两个 step，而整体的方法只前进一个 step。SGD 方法的一个缺点是，其更新方向完全依赖于当前的 batch，因而其更新十分不稳定。Momentum 即动量，它模拟的是物体运动时的惯性，即更新的时候在一定程度上保留之前更新的方向，同时利用当前 batch 的梯度微调最终的更新方向。这样一来，可以在一定程度上增加稳定性，从而学习得更快。

```
loss_function = nn.CrossEntropyLoss()
```

```
    #损失函数采用(输出，目标)输入对，并计算估计输出距离目标距离的值。nn包下
有几种不同的损失函数，具体参考官网提供的损失函数说明。一个简单的损失
是:nn.MSELoss，用于计算输入和目标之间的平均平方误差。
    for epoch in range(EPOCH):
        for step, (x, y) in enumerate(train_loader):
            b_x = Variable(x)
            b_y = Variable(y)

            output = cnn(b_x)
            loss = loss_function(output, b_y)
            optimizer.zero_grad()

            loss.backward()
            optimizer.step()

    #指定optimizer，loss function，需要特别指出的是记得每次反向传播前都要
清空上一次的梯度，optimizer.zero_grad()。
            if step % 100 == 0:
                test_output = cnn(test_x)
                pred_y = torch.max(test_output, 1)[1].data.squeeze()
                accuracy = sum(pred_y == test_y) / test_y.size(0)
                print('Epoch:', epoch, '|Step:', step,
                      '|train loss:%.4f'%loss.data[0], '|test
accuracy:%.4f'%accuracy)

    test_output = cnn(test_x[:20])
    pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
    print(pred_ y[:20], 'prediction number')
    print(test_y[:20].numpy(), 'real number')
```

输出结果，从输出结果来看，卷积神经网络预测效果还是不错的。从上面输出的训练次数、误差大小，以及精度来看，最后的训练精度达到 99%。通过训练我们可以看出卷积神经网络模型应用在 MNIST 效果上还是很不错的。

```
    '''''
    Epoch:  0 | train loss: 2.3145 | test accuracy: 0.17
    Epoch:  0 | train loss: 0.5858 | test accuracy: 0.91
```

```
○○○○○○○○
Epoch:  2 | train loss: 0.0168 | test accuracy: 0.99
Epoch:  2 | train loss: 0.0091 | test accuracy: 0.99
Epoch:  2 | train loss: 0.0301 | test accuracy: 0.99
Epoch:  2 | train loss: 0.0237 | test accuracy: 0.99
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4] prediction number
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4] real number
```

从预测手写数字的结果来看，预测值与真实值基本吻合。

7.5 深度残差模型案例

Resnet 在 ILSVRC 和 COCO 2015 的五个主要任务轨迹中都获得了第一名的成绩。

ImageNet 分类任务："超级深"的 152 层网络。

ImageNet 检测任务：超过第二名 16%。

ImageNet 定位任务：超过第二名 27%。

COCO 检测任务：超过第二名 11%。

COCO 分割任务：超过第二名 12%。

下面我们用实例来实现残差网络模型。

本例文件名为 PyTorch/Chapter07/pt2_deep_residual_network.py。

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
#导入模块包

transform = transforms.Compose([
    transforms.Scale(40),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
```

```
    transforms.ToTensor()])
#图形变换
常用的图形变换如下：
torchvision.transforms.Compose(transforms)
#将多个 transform 组合起来使用。
 torchvision.transforms.Scale(size, interpolation=2)
#按照规定的尺寸重新调节 PIL.Image。
torchvision.transforms.RandomHorizontalFlip ( )
#随机水平翻转给定的 PIL.Image,概率为 0.5。
 torchvision.transforms.RandomCrop(size, padding=0)
#切割中心点的位置随机选取。



train_dataset = dsets.CIFAR10(root='./data/',
                          train=True,
                          transform=transform,
                          download=True)

test_dataset = dsets.CIFAR10(root='./data/',
                         train=False,
                         transform=transforms.ToTensor())
```

本节使用的是比较经典的数据集叫作 CIFAR-10，包含 60000 张 32×32 的彩色图像，因为是彩色图像，所以这个数据集是三通道的，分别是 R、G、B 三个通道。CIFAR-10，一共有 10 类图片，每一类图片有 6000 张，有飞机、鸟、猫、狗等，而且其中没有任何重叠的情况。现在还有一个版本，CIFAR-100，里面有 100 类。这里还要提到一个数据增广的问题，对于数据集比较小，数据量远远不够的情况，我们可以对图片进行翻转、随机剪切等操作来增加数据，制造出更多的样本，提高对图片的利用率。

```
    train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
                                       batch_size=100,
                                       shuffle=False)

    test_loader =
torch.utils.data.DataLoader(dataset=test_dataset,
                                      batch_size=100,
```

```
                                              shuffle=False)
    #加载数据
    def conv3x3(in_channels, out_channels, stride=1):
        return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                         stride=stride, padding=1, bias=False)
    #nn.conv1d(input, weight, bias=None, stride=1, padding=0,
dilation=1, groups=1)
```

对输入的数据进行二维卷积。卷积的本质就是用卷积核的参数来提取原始数据的特征，通过矩阵点乘的运算，提取出和卷积核特征一致的值，如果卷积层有多个卷积核，则神经网络会自动学习卷积核的参数值，使得每个卷积核代表一个特征。

参数说明如下。

input：输入的 Tensor 数据，格式为（batch，channels，W），三维数组，第一维度是样本数量，第二维度是通道数或者记录数，第三维度是宽度。

weight：过滤器，也叫卷积核权重。是一个三维数组，（out_channels，in_channels/groups，kW）。out_channels 是卷积核输出层的神经元个数，也就是这层有多少个卷积核；in_channels 是输入通道数；kW 是卷积核的宽度。

bias：位移参数。

stride：滑动窗口，默认为 1，指每次卷积对原数据滑动 1 个单元格。

padding：是否对输入数据填充 0。padding 可以将输入数据的区域改造成卷积核大小的整数倍，这样对不满足卷积核大小的部分数据就不会忽略了。通过 padding 参数指定填充区域的高度和宽度，默认为 0。

dilation：卷积核之间的空格，默认为 1。

groups：将输入数据分成组，in_channels 应该被组数整除，默认为 1。

Conv2d 是二维卷积，它和 conv1d 的区别在于对宽度进行卷积，对高度进行卷积，而一维卷积对高度不进行卷积。

```
    class ResidualBlock(nn.Module):
        def __init__(self, in_channels, out_channels, stride=1,
downsample=None):
```

```
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample
    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

残差结构，深度网络容易造成梯度在 back propagation 的过程中消失，导致训练效果很差，而深度残差网络在神经网络的结构层面解决了这一问题，使得就算网络很深，梯度也不会消失。使用预激活残差单元构筑的残差网络，相较于使用原始单元更易收敛，且有一定正则化的效果，测试集上性能也普遍好于原始残差单元。

每个 Convx_x 中都含有 3 个残差模块，每个模块的卷积核都是 3×3 大小的，pad 为 1，stride 为 1。Con4_x 的输出通过 global_average_pooling 映射到 64 个 1×1 大小的特征图上，最后再通过含有 10 个神经元的全连接层输出分类结果。

```
# ResNet 模型
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
```

```
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[0], 2)
        self.layer3 = self.make_layer(block, 64, layers[1], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, num_classes)
```

一个残差模块是由两层卷积再加一个恒等映射组成的。特征图的大小是一样的，残差模块的输入输出的维度大小也是一样的，可以直接进行相加。

```
       def make_layer(self, block, out_channels, blocks,
stride=1):
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels,
stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels,
stride, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)
    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

resnet = ResNet(ResidualBlock, [3, 3, 3])
```

```
resnet.cuda()
```

网络的层数越深，可覆盖的解空间越广，理论上应该精度越高。

但简单地累加层数，并不能直接带来更好的收敛性和精度。

```
criterion = nn.CrossEntropyLoss()
lr = 0.001
optimizer = torch.optim.Adam(resnet.parameters(), lr=lr)
```

损失和优化函数，学习速率为 0.001，如果选择较大的学习速率会导致训练速度过快，导致效果不好。如果学习速率过小，导致训练速度太慢，所以选择合适的学习速率可以提高效果和缩短时间。

```
for epoch in range(80):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images.cuda())
        labels = Variable(labels.cuda())

        optimizer.zero_grad()
        outputs = resnet(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

进行训练时，总共训练 80 批次，同时把图片数据转换成 PyTorch 可识别的变量，利用 CUDA 进行加速运算。

```
        if (i+1) % 100 == 0:
            print ("Epoch [%d/%d], Iter [%d/%d] Loss: %.4f"
%(epoch+1, 80, i+1, 500, loss.data[0]))

    if (epoch+1) % 20 == 0:
        lr /= 3
        optimizer = torch.optim.Adam(resnet.parameters(), lr=lr)
       #每隔 100 次，输出一次结果

correct = 0
total = 0
```

```
    for images, labels in test_loader:
        images = Variable(images.cuda())
        outputs = resnet(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
    correct += (predicted.cpu() == labels).sum()
    # 测试结果

    print('Accuracy of the model on the test images: %d %%' % (100
* correct / total))
    #输出精度

    torch.save(resnet.state_dict(), 'resnet.pkl')

    # 保存模型
```

我们从训练的模型结果来看，经过训练，模型的精确度达到 99%。我们可以进一步增加训练次数，或者增加需要训练的图像数量来进行训练。

```
    Epoch [76/80], lter [100/500], Loss: 0.9914
    Epoch [77/80], lter [200/500], Loss: 0.9014
    Epoch [78/80], lter [300/500], Loss: 0.8524
    Epoch [79/80], lter [400/500], Loss: 0.8746
    Epoch [90/80], lter [500/500], Loss: 0.8463
    Test Accuracy of the model on the test images: 99 %
```

第 8 章

8

循环神经网络简介

循环神经网络（Recurrent Neural Network，RNN）源自 1982 年由 Saratha Sathasivam 提出的霍普菲尔德网络。

霍普菲尔德网络因为实现困难，该网络结构也于 1986 年后被全连接神经网络以及一些传统的机器学习算法所取代。随着更加有效的循环神经网络结构被不断提出，循环神经网络挖掘数据中的时序信息以及语义信息的深度表达能力被充分利用，循环神经网络的主要用途是处理和预测序列数据。擅长解决序列化相关问题。包括不限于序列化标注问题、NER、POS、语音识别等。它已经在众多自然语言处理（Natural Language Processing，NLP）中取得了巨大成功并被广泛应用于语音识别、手写体识别等。目前有很多人工智能应用都依赖于循环神经网络，在谷歌（语音搜索）、百度（Deep Speech）和亚马逊的产品中都能看到 RNN 的身影。

循环神经网络是一个在时间上传递的神经网络，网络的深度就是时间的长度。该神经网络是专门用来处理时间序列问题的，能够提取时间序列的信息。与传统的神经网络不同，RNN 能利用“序列信息”。传统的神经网络模型中，我们假设所有的输入是相互独立的。从输入层到隐藏层再到输出层，层与层之间是全连接的，每层之间的节点是无连接的。循环神经网络与传统的前馈神经网络有所不同，循环神经网络的隐藏层相互连接，即一个序列当前的输出与前面的输出也有关。循环神经网络会对于每一个

时刻的输入结合当前模型的状态给出一个输出。RNN 对序列的每个元素执行同样的操作，其输出依赖于前次计算的结果。RNN 引入了定向循环，能够处理那些输入之间前后关联的问题。RNN 拥有捕获已计算节点信息的记忆能力。

8.1 循环神经网络模型结构

图 8.1 是一个简单的循环神经网络，循环体中的神经网络的输入有两部分，一部分为上一时刻的状态，另一部分为当前时刻的输入样本。它由输入层、一个隐藏层和一个输出层组成。

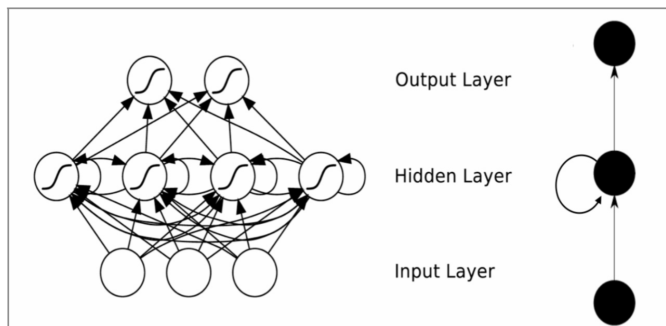


图 8.1 循环神经网络结构模型

x 是输入层的值，s 是隐藏层的值，U 是输入层到隐藏层的权重矩阵。o 表示输出层的值，V 是隐藏层到输出层的权重矩阵。循环神经网络的隐藏层的值 s 不仅仅取决于当前这次的输入 x，还取决于上一次隐藏层的值 s。权重矩阵 W 就是隐藏层上一次的值作为这一次的输入的权重。对于一个序列数据，可以将这个序列上不同时刻的数据依次传入循环神经网络的输入层，而输出可以是对序列中下一个时刻的预测，也可以是对当前时刻信息的处理结果。如在机器翻译的任务中，对于源语言中的每个词向量，网络可以精准输出目标语言中的单词。

循环神经网络内部结构展开如图 8.2 所示。

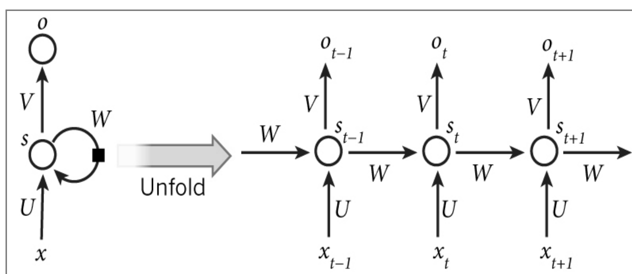


图 8.2　循环神经网络展开图

RNN 是根据下面的步骤来迭代和做计算的。

x_t 表示 t=1,2,3…步的输入。

s_t 是隐层第 t 步的状态，s_t 由当前步骤的输入和先前隐藏状态共同计算求得，$S_t = f(Ux_t + Ws_t - 1)$。

O_t 是第 t 步的输出。

循环神经网络的结构特征可以很容易得出它最擅长解决的问题是与时间序列相关的。循环神经网络也是处理这类问题时最自然的神经网络结构。

循环神经网络中由于输入时叠加了之前的信号，所以反向传导时不同于传统的神经网络，因为对于时刻 t 的输入层，其残差不仅来自输出，还来自之后的隐藏层。通过反向传递算法，利用输出层的误差，求解各个权重的梯度，然后利用梯度下降法更新各个权重。它的展开图中信息流向也是确定的，没有环流，循环神经网络是时间维度上的深度模型，可以对序列内容建模。但是需要训练的参数较多，容易出现梯度消散或梯度爆炸的问题，不具有特征学习能力。

RNN 已经在实践中被证明对 NLP 是非常成功的。如词向量表达、语句合法性检查、词性标注等。

8.2　不同类型的RNN

1．Simple RNN（SRN）

SRN 是 RNN 的一种特例，它是一个三层网络，并且在隐藏层增加了

上下文单元。上下文单元节点与隐藏层中的节点的连接是固定(谁与谁连接)的，并且权值也是固定的（值是多少）。在每一步中，使用标准的前向反馈进行传播，然后使用学习算法进行学习。上下文每一个节点保存其连接的隐藏层节点的上一步的输出，即保存上文；并作用于当前步对应的隐藏层节点的状态，即隐藏层的输入由输入层的输出与上一步自己的状态所决定。因此 SRN 能够解决标准的多层感知机（MLP）无法解决对序列数据进行预测的任务。

2．Bidirectional RNN

Bidirectional RNN 是双向 RNN，当前的输出和之前的序列元素，以及之后的序列元素都是有关系的。例如：预测一个语句中缺失的词语就需要根据上下文来进行预测。Bidirectional RNN 是一个相对较简单的 RNN，是由两个 RNN 上下叠加在一起组成的。输出是由这两个 RNN 的隐藏层的状态决定的，如图 8.3 所示。

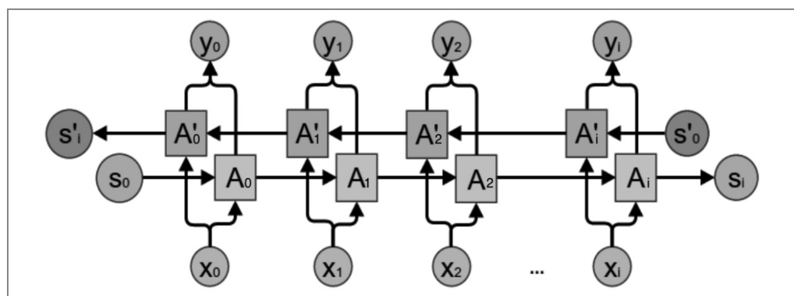


图 8.3　Bidirectional RNN 模型

3．深层双向 RNN

深层双向 RNN 和双向 RNN 比较类似，区别只是每一步/每个时间点设定为多层结构，如图 8.4 所示。

4．LSTM 神经网络

Long Short Term 网络叫作 LSTM，是一种 RNN 特殊的类型。LSTM 由 Hochreiter & Schmidhuber（1997）提出，并被 Alex Graves 进行了改良和推广。LSTM 精确解决了 RNN 的长短记忆问题。

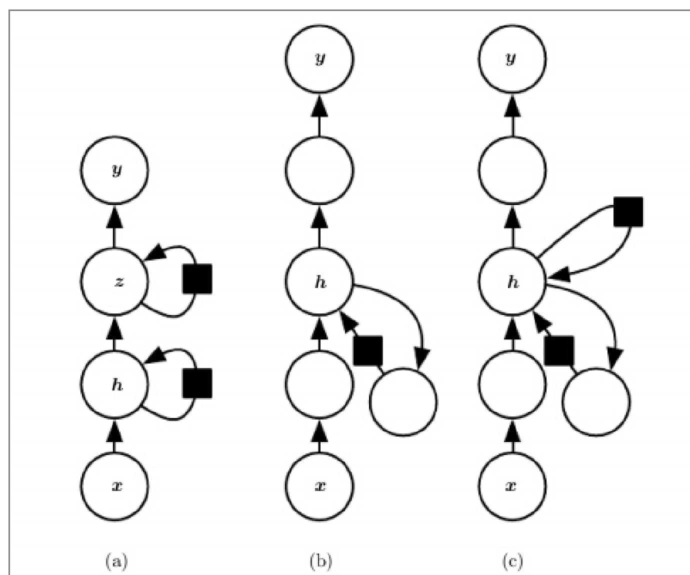


图 8.4　深层双向 RNN

在 LSTM 中，每个神经元是一个“记忆细胞”，细胞里面有一个“输入门”（input gate），一个“遗忘门”（forget gate），一个“输出门”（output gate），俗称“三重门”。与一般神经网络的神经元相比，LSTM 神经元多了一个遗忘门。结构如图 8.5 所示。

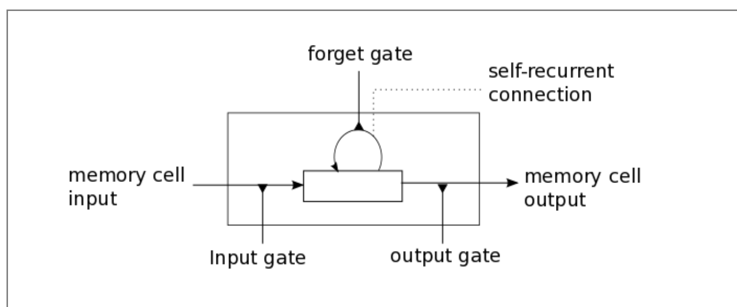


图 8.5　LSTM 神经网络结构图

它与一般的 RNN 结构在本质上并没有什么不同，只是使用了不同的函数去计算隐藏层的状态，如图 8.6 所示。在 LSTM 中，i 结构被称为 cells，可以把 cells 看作是黑盒，用以保存当前输入 x_t 之前保存的状态 h_{t-1}，这些 cells 以一定的条件决定哪些 cell 抑制，哪些 cell 兴奋。它们可以结合前面

的状态、当前的记忆与当前的输入。该网络结构在对长序列依赖问题中非常有效。

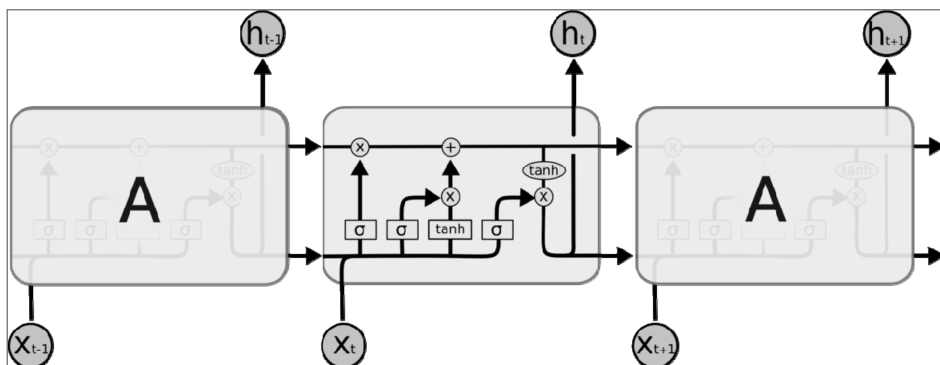


图 8.6　LSTM 内部结构

LSTM 神经元的输出除了与当前输入有关外，还与自身记忆有关。RNN 的训练算法也是基于传统 BP 算法，并且增加了时间考量，称为 BPTT（Back-propagation Through Time）算法。Google 推出的邮件智能回复也是基于 LSTM 模型，只不过是用了一对，一个用来编码邮件，一个用来解码回复。

图 8.7 是各种元素的图标。

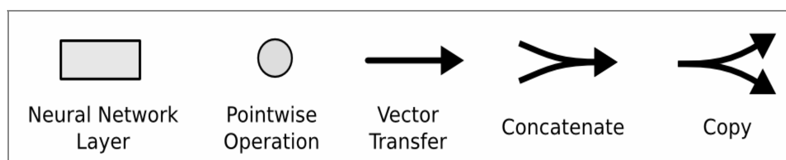


图 8.7　LSTM 元素的图标

Neural NetWork Layer：表示一个神经网络层。

Pointwise Operation：表示一种数学操作。

Vector Tansfer：表示每条线代表一个向量，从一个节点输出到另一个节点。

Concatenate：表示两个向量的合并。

Copy：表示复制一个向量变成相同的两个向量。

8.3　LSTM结构具体解析

我们先理解门（gates）的结构。

输入门 i_t：控制有多少信息可以流入记忆细胞。

遗忘门 f_t：控制有多少上一时刻的记忆细胞中的信息可以累积到当前时刻的记忆细胞中，如图 8.8 所示。

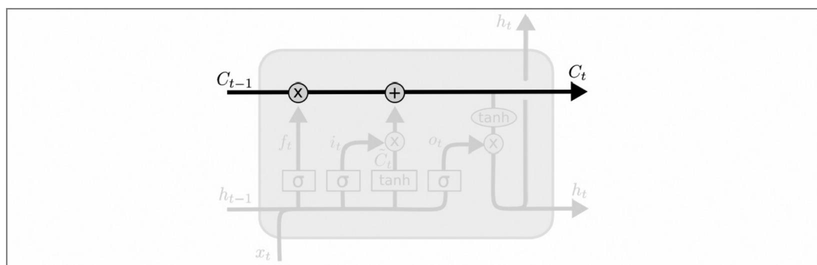


图 8.8　LSTM 的遗忘门

输出门 o_t：控制有多少当前时刻的记忆细胞中的信息可以流入当前隐藏状态 h_t 中。

LSTM 的关键就是细胞核的状态。细胞核的状态类似于一种传送带。它直接在整个链上穿过，附带一些少量的线性交互，让信息在上面流传而保持不变。

图 8.9 是通过一种叫作门（gates）的结构，让信息有选择性地通过。它们是由一个 Sigmoid 神经网络层和一个 Pointwise 的乘法操作组成的。

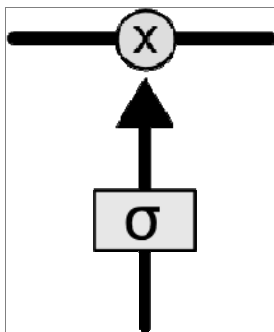


图 8.9　遗忘门的内部结构

第一步：利用遗忘门层，决定从细胞状态中丢弃什么信息，衰减系数计算如图 8.10 所示。读取 $h_{t\text{-}1}$ 和 x_t，输出一个在 0 到 1 之间的数值给每个在细胞状态 $C_{t\text{-}1}$ 中的数字。这决定我们会从细胞状态中丢弃什么信息。由于 Sigmoid 输出结果为 0 到 1，所以用 1 表示“完全保留”，0 表示“完全舍弃”。

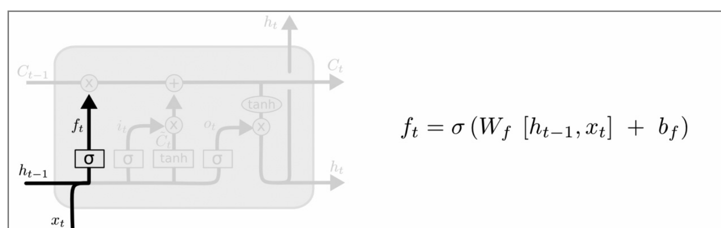


图 8.10　衰减系数计算过程

第二步：更新信息。首先，Sigmoid 层为“输入门层”，决定什么值我们将要更新。然后，tanh 层创建一个新的候选值向量。计算过程如图 8.11 所示。

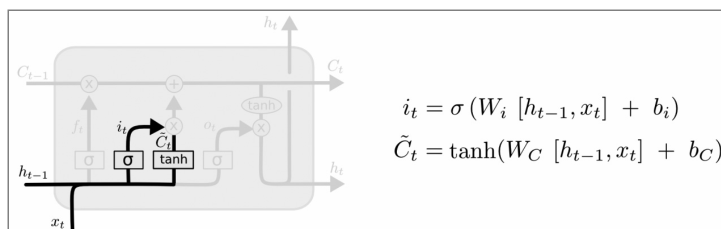


图 8.11　*t* 时刻的记忆计算过程

第三步：更新旧细胞状态的时间，$C_{t\text{-}1}$ 更新为 C_t。计算过程如图 8.12 所示。

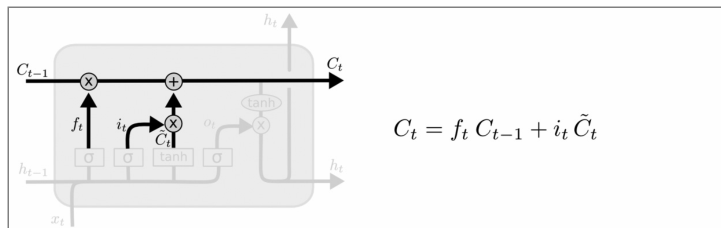


图 8.12　$C_{t\text{-}1}$ 更新为 C_t 过程

第四步：输出门，确定输出什么值。此时的输出是根据上述第三步的 C_t 状态进行计算的。计算过程如图 8.13 所示。

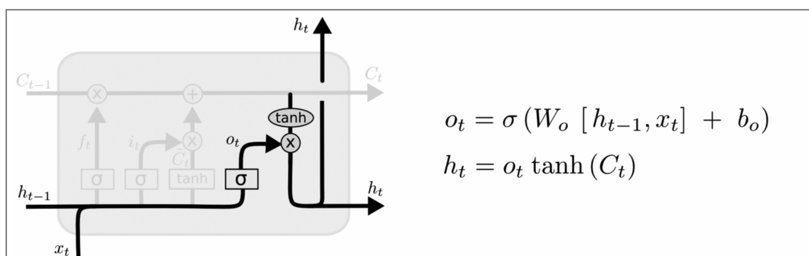


图 8.13　确定输出值

汇总之后展开公式如图 8.14 所示。

$$i_t = sigmoid(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = sigmoid(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

$$o_t = sigmoid(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

$$c_t = f_t c_{t\text{-}1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

$$h_t = o_t tanh(\mathrm{c_t})$$

图 8.14　LSTM 计算流程

8.4　LSTM的变体

1. GRN

Gated Recurrent Unit，也称为 GRU，由 Cho 等人（2014）提出，是 LSTM 的变体。遗忘门和输入门结合作为“更新门”（update gate），同时还做了其他的一些改变。与 RNN 不同的是，序列中不同的位置的单词对当前隐藏层的状态的影响不同，越靠前面的影响越小，即每个前面状态对当前的影响进行了距离加权，距离越远，权值越小。另外，在产生误差 error 时，误差可能是由某一个或者几个单词引发的，所以应当仅仅对对应的单词 weight 进行更新。GRU 的结构如图 8.15 所示。GRU 首先根据当前输入单词向量 word vector 在前一个隐藏层的状态中计算出 update gate 和 reset gate。再根

据 reset gate、当前 word vector 以及前一个隐藏层计算新的记忆单元内容（New Memory Content）。当 reset gate 为 1 的时候，前一个隐藏层计算新的记忆单元内容忽略之前的所有记忆单元内容，最终的记忆是之前的隐藏层与新的记忆单元内容的结合。

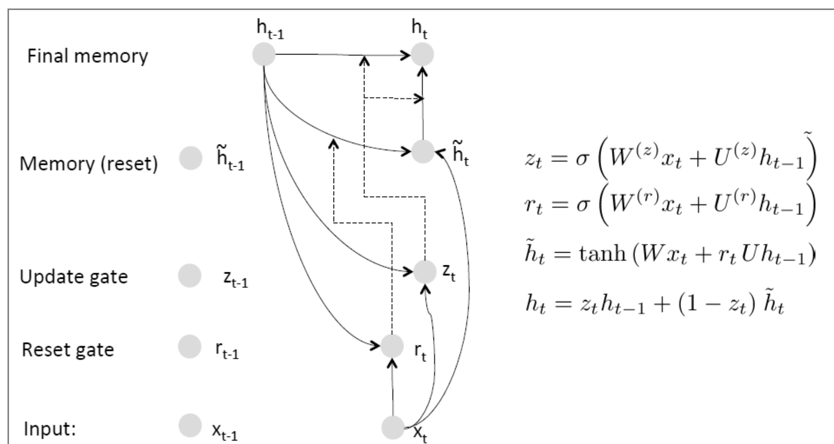


图 8.15　GRU 结构图

2．CW-RNN

CW-RNN 是较新的一种 RNN 模型，其论文发表于 2014 年 Beijing ICML。是一种使用时钟频率来驱动的 RNN。它将隐藏层分为几组，每一组按照自己规定的时钟频率对输入进行处理。并且为了降低标准 RNN 的复杂性，CW-RNN 减少了参数的数目，提高了网络性能，加快了网络的训练速度。CW-RNN 通过不同的隐藏层模块工作，在不同的时钟频率下来解决长时间依赖问题。将时钟时间进行离散化，然后在不同的时间点，不同的隐藏层组中工作。因此，所有的隐藏层组不会在每一步都同时工作，这样便会加快网络的训练。并且，时钟周期短的组的神经元不会连接到时钟周期长的组的神经元上，只会让周期长的连接到周期短的上，周期长的速度慢，周期短的速度快，那么便是速度慢的连速度快的。

CW-RNN 包括输入层、隐藏层、输出层。输入层到隐藏层的连接，隐藏层到输出层的连接为前向连接，如图 8.16 所示。

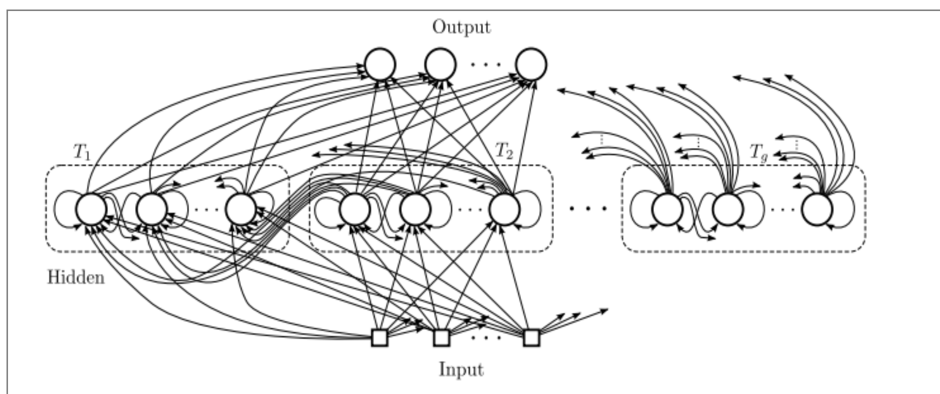


图 8.16　CW-RNN 结构图

3．RNN 的应用

（1）机器翻译（Machine Transtation）

机器翻译是将一种源语言语句变成意思相同的另一种源语言语句，如将英语语句变成同样意思的中文语句。与语言模型关键的区别在于，需要将源语言语句序列输入后，才进行输出，即输出第一个单词时，便需要从完整的输入序列中进行获取。机器翻译如图 8.17 所示。

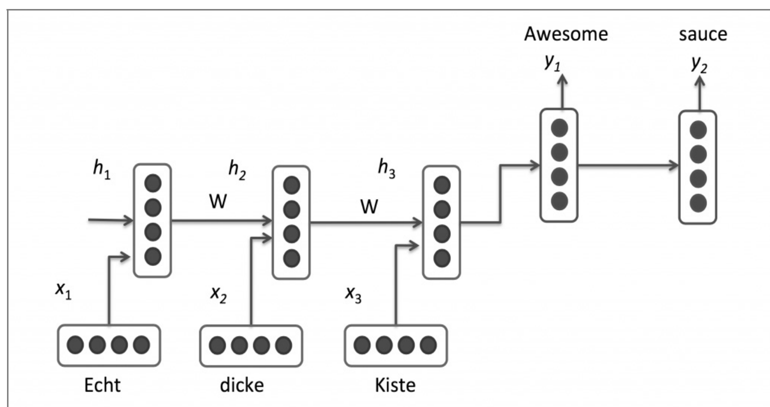


图 8.17　机器翻译示例

（2）语音识别（Speech Recognition）

语音识别是指给一段声波的声音信号，预测该声波对应的某种指定源语言的语句以及该语句的概率值。

（3）图像描述生成（Generating Image Descriptions）

RNN 已经在对无标图像描述自动生成中得到应用。将 CNN 与 RNN 结合进行图像描述自动生成，这是一个非常神奇的研究与应用。该组合模型能够根据图像的特征生成描述，如图 8.18 所示。



图 8.18　CNN 与 RNN 结合自动生成图像描述

8.5　循环神经网络实现

8.5.1　循环神经网络案例

前面我们已经学习了循环神经网络的基本原理，下面我们用具体的案例来实现。

本例文件名为 PyTorch/Chapter09/pt1_recurrent_neural_networks.py。

```
#加载所需的模块包
import torch
```

```
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable


#设置参数
sequence_length = 28
input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10
batch_size = 100
num_epochs = 2
learning_rate = 0.01
```

下载数据 MNIST 数据集来自美国国家标准与技术研究所，National Institute of Standards and Technology（NIST）。训练集（Training set）由来自 250 个不同人手写的数字构成，其中 50%是高中学生，50%来自人口普查局（the Census Bureau）的工作人员。测试集（Test set）也是同样比例的手写数字数据。MNIST 数据集可在 http://yann.lecun.com/exdb/mnist/获取，它包含了四个部分。

Training set images：train-images-idx3-ubyte.gz（9.9 MB，解压后 47 MB，包含 60000 个样本）

Training set labels：train-labels-idx1-ubyte.gz（29 KB，解压后 60 KB，包含 60000 个标签）

Test set images：t10k-images-idx3-ubyte.gz（1.6 MB，解压后 7.8 MB，包含 10000 个样本）

Test set labels：t10k-labels-idx1-ubyte.gz（5 KB，解压后 10 KB，包含 10000 个标签）

```
#数据预处理
train_dataset = dsets.MNIST(root='./data/',
                            train=True,
                            transform=transforms.ToTensor(),
```

```
                                download=True)

    test_dataset = dsets.MNIST(root='./data/',
                               train=False,
                               transform=transforms.ToTensor())
```

参数说明如下。

- root：数据集。
- train：True＝ 训练集，False＝ 测试集。
- download：如果为 True，请从 Internet 下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。
- transform：将原数据规范化到（0，1）区间，数据变换（0，255）≥（0，1）。
- 由于加载的数据集为 array 格式，需要转换成 Torch 能识别的 Tensor，利用 torchvision.datasets 加载 DataLoader 中的 dataset 格式的数据，同时在获取的时候直接转换成训练所需的数据格式。

```
    #加载数据
    train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

    test_loader =
torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

    #定义循环神经网络结构
    class RNN(nn.Module):
        def __init__(self, input_size, hidden_size, num_layers,
num_classes):
            super(RNN, self).__init__()
            self.hidden_size = hidden_size
            self.num_layers = num_layers
            self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
```

```
        self.fc = nn.Linear(hidden_size, num_classes)
```

torch.nn.LSTM（input_size, hidden_size, num_layers, batch_first=True）将一个多层的（LSTM）应用到输入序列。

参数说明如下。

input_size：输入的特征维度。

hidden_size：隐状态的特征维度。

num_layers：层数。

bias：如果为 False，那么 LSTM 将不会使用b_{ih},b_{hh}，默认为 True。

batch_first：如果为 True，那么输入和输出 Tensor 的形状为（batch，seq，feature）。

dropout：如果非零的话，将会在 RNN 的输出上加个 dropout，最后一层除外。

Bidirectional：如果为 True，将会变成一个双向 RNN，默认为 False。

循环神经网络工作的关键点就是使用历史信息来帮助当前的决策。LSTM 靠一些“门”的结构让信息有选择性地影响每个时刻循环神经网络中的状态。

```
    #加载数据
    train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
                                              batch_size=batch_size,
                                              shuffle=True)

    test_loader =
torch.utils.data.DataLoader(dataset=test_dataset,
                                             batch_size=batch_size,
                                             shuffle=False)

    #定义循环神经网络结构
    class RNN(nn.Module):
        def __init__(self, input_size, hidden_size, num_layers,
```

```
num_classes):
          super(RNN, self).__init__()
          self.hidden_size = hidden_size
          self.num_layers = num_layers
          self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
          self.fc = nn.Linear(hidden_size, num_classes)
```

从输出结果来看，精度为 92%，说明循环神经网络在 MNIST 中，效果良好。

8.5.2 双向 RNN 案例

前面我们已经简单介绍了 Bidirectional RNN 原理，下面我们用代码来实现。

本例文件名为 PyTorch/Chapter09/pt2_recurrent_neural_networks.py。

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
# 导入模块

sequence_length = 28
input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10
batch_size = 100
num_epochs = 2
learning_rate = 0.003
#设置参数

train_dataset = dsets.MNIST(root='./data/',
                       train=True,
                       transform=transforms.ToTensor(),
```

```
                                download=True)

    test_dataset = dsets.MNIST(root='./data/',
                               train=False,
                               transform=transforms.ToTensor())
    train_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
                                            batch_size=batch_size,
                                            shuffle=True)

    test_loader =
torch.utils.data.DataLoader(dataset=test_dataset,
                                           batch_size=batch_size,
                                           shuffle=False)
    # 加载数据

    #定义双向循环网络模型
    class BiRNN(nn.Module):
        def __init__(self, input_size, hidden_size, num_layers,
num_classes):
            super(BiRNN, self).__init__()
            self.hidden_size = hidden_size
            self.num_layers = num_layers
            self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
                           batch_first=True, bidirectional=True)
            self.fc = nn.Linear(hidden_size*2, num_classes)

        def forward(self, x):
            h0 = Variable(torch.zeros(self.num_layers*2, x.size(0),
self.hidden_size)).cuda()
            c0 = Variable(torch.zeros(self.num_layers*2, x.size(0),
self.hidden_size)).cuda()

            out, _ = self.lstm(x, (h0, c0))

            out = self.fc(out[:, -1, :])
            return out
```

```
    rnn = BiRNN(input_size, hidden_size, num_layers, num_classes)
    rnn.cuda()
```

由于标准的循环神经网络（RNN）在时序上处理序列，往往忽略了未来的上下文信息。一种很显而易见的解决办法是在输入和目标之间添加延迟，进而可以给网络一些时步来加入未来的上下文信息，双向循环神经网络（BRNN）的基本思想是每一个训练序列向前和向后分别是两个循环神经网络（RNN），而且这两个都连接着一个输出层。这个结构提供给输出层输入序列中每一个点的完整的过去和未来的上下文信息。输入在向前和向后隐含层，隐含层到隐含层，向前和向后隐含层到输出层之中。每一个时刻都在重复利用。值得注意的是：向前和向后隐含层之间没有信息流，这保证了展开图是非循环的。

```
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam（rnn.parameters(),
lr=learning_rate）
```

损失和优化函数，损失函数采用交叉熵损失函数，交叉熵可在神经网络中作为损失函数，p 表示真实标记的分布，q 则为训练后模型的预测标记分布，交叉熵损失函数可以衡量 p 与 q 的相似性。交叉熵作为损失函数还有一个好处是使用 Sigmoid 函数在梯度下降时能避免均方误差损失函数学习速率降低的问题，因为学习速率可以被输出的误差所控制。

```
    #训练模型
    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(train_loader):
            images = Variable(images.view(-1, sequence_length,
input_size)).cuda()
            labels = Variable(labels).cuda()

            optimizer.zero_grad()
            outputs = rnn(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            if (i+1) % 100 == 0:
                print ('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'
```

```
                    %(epoch+1, num_epochs, i+1,
len(train_dataset)//batch_size, loss.data[0]))

    # 对模型进行测试
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = Variable(images.view(-1, sequence_length,
input_size)).cuda()
        outputs = rnn(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted.cpu() == labels).sum()
    #输出结果
    print('Test Accuracy of the model on the 10000 test images: %d
%%' % (100 * correct / total))

    # 保存模型
    torch.save(rnn.state_dict(), 'rnn.pkl')

    #输出结果为
    Epoch [1/2], Step [100/600], Loss: 0.4944
    Epoch [1/2], Step [200/600], Loss: 0.3014
    Epoch [1/2], Step [300/600], Loss: 0.1435
    Epoch [1/2], Step [400/600], Loss: 0.1359
    Epoch [1/2], Step [500/600], Loss: 0.2384
    Epoch [1/2], Step [600/600], Loss: 0.0595
    Epoch [2/2], Step [100/600], Loss: 0.1450
    Epoch [2/2], Step [200/600], Loss: 0.1012
    Epoch [2/2], Step [300/600], Loss: 0.0836
    Epoch [2/2], Step [400/600], Loss: 0.0916
    Epoch [2/2], Step [500/600], Loss: 0.1639
    Epoch [2/2], Step [600/600], Loss: 0.0493
    Test Accuracy of the model on the 10000 test images: 98 %
```

从输出的结果来看，双向循环网络模型经过训练之后，对图片进行训练的精度为 98%，说明双向循环神经网络效果还是很不错的。

9

第 9 章

自编码模型

在有监督学习中，训练样本是有类别标签的。现在假设我们只有一个没有带类别标签的训练样本集合 $x^{(1)},x^{(2)},x^{(3)},\ldots$其中$x^{(i)}\in \mathrm{R}^{n}$。其中的一种算法叫作 AutoEncoder——自编码网络。自编码神经网络是一种无监督学习算法，它使用了反向传播算法，并让目标值等于输入值，简单的自编码是一种三层神经网络模型，包含了数据输入层、隐藏层、输出重构层，同时它是一种无监督学习模型。在有监督的神经网络中，我们的每个训练样本是（x，y），y 一般是我们人工标注的数据。比如我们用于手写的字体分类，那么 y 的取值就是 0～9 之间的数值，最后神经网络设计的时候，网络的输出层是一个 10 个神经元的网络模型（比如网络输出是（0，0，1，0，0，0，0，0，0，0），那么就表示该样本标签为 2）。然而自编码是一种无监督学习模型，我们训练数据本来是没有标签的，那么自编码是这样干的，它令每个样本的标签为 $y=x$，也就是每个样本的数据 x 的标签也是 x。自编码就相当于自己生成标签，而且标签是样本数据本身。图 9.1 是一个自编码神经网络的示例。

如图 9.2 所示，网络中最左侧节点是输入层，最右侧一列神经元是输出层。

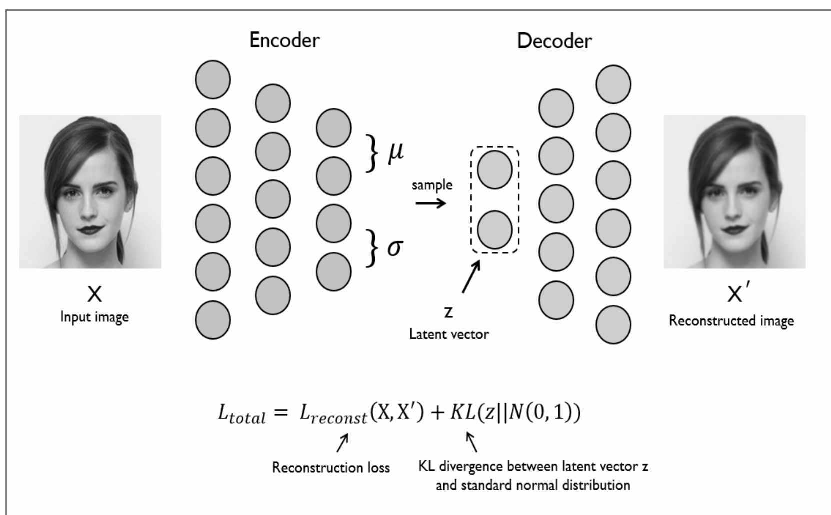


图 9.1　自编码模型

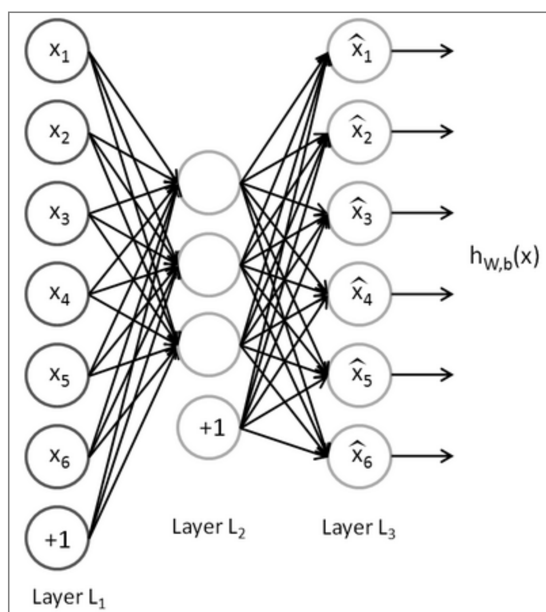


图 9.2　自编码模型内部结构

输出层的神经元数量完全等于输入层神经元的数量。隐藏层的神经元数量少于输出层。自编码网络的作用是将输入样本压缩到隐藏层，再在输出端重建样本，即压缩和解压。自编码神经网络尝试 $h_{W,b}(x) \approx x$ 。

自编码网络输出层与输入层存在如下关系：$\hat{x}_i \approx x_i$ 。

自编码网络是将经过压缩的数据还原，在压缩的过程中，限制隐藏层的稀疏性。神经元总是使用一个激活函数，神经元分为“激活状态”和“非激活状态”，如果大部分神经元处于非激活状态，这时候隐藏层中的激活神经元是“稀疏”的，即稀疏性。然后目标函数为了还原数据应该使得损失尽量小，定义如下：

$$J(W,b)=\frac{1}{m}\sum_{i=1}^{m}(\hat{x}-x)^2$$

如果我们输入一张 10×10 像素的灰度图像，这样就有 100 个像素点，所以输入层和输出层的节点数量就是 100，而我们取隐藏层节点数量为 25。要求每一个输出神经元的输出值和输入图像的对应像素灰度相同，这样就会迫使隐藏层节点学习得到输入数据的压缩表示方法，迫使隐藏层要用 25 维数据重构出 100 维的数据，这样也就完成了学习压缩过程。

通常隐含层的神经元数目要比输入/输出层的少，这样做的目的是为了神经网络只学习最重要的特征并实现特征降维。

它能从原数据中总结出每种类型数据的特征，如果把这些特征类型都放在一张二维的图片上，每种类型都已经被很好地用原数据的精髓区分开来。如果你了解 PCA 主成分分析，再提取主要特征时，自编码和它一样，甚至超越了 PCA。换句话说，自编码可以像 PCA 一样给特征属性降维。

接下来学习自编码实现。

上面，我们已经学习了自编码的原理，下面用代码来实现。

本例文件名为 PyTorch/Chapter10/pt1_variational_auto_encoder.py。

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.utils.data as Data
import torchvision
import matplotlib.pyplot as plt
import numpy as np
#导入各种需要的pytorch包
```

```
torch.manual_seed(1)


EPOCH = 10
BATCH_SIZE = 64
LR = 0.005
N_TEST_IMG = 5
#设置各种参数

train_data = torchvision.datasets.MNIST(
    root='./mnist/',
    train=True,
    transform=torchvision.transforms.ToTensor(),
    download=DOWNLOAD_MNIST,
)
#dset.MNIST(root, train=True, transform=None,
target_transform=None, download=False)
```

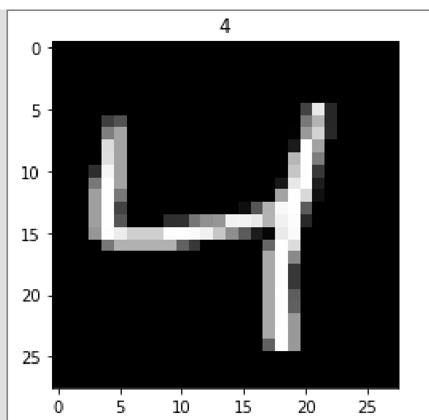
参数说明如下。

root：processed/training.pt 和 processed/test.pt 的主目录。

train：True = 训练集，False = 测试集。

download：如果为 True，从互联网上下载数据集，并把数据集放在 root 目录下。如果数据集之前下载过，将处理过的数据（minist.py 中有相关函数）放在 processed 文件夹下。

```
# plot one example
print(train_data.train_data.size())
#大小为 (60000, 28, 28)
print(train_data.train_labels.size())
# 标签大小为(60000)
plt.imshow(train_data.train_data[2].numpy(), cmap='gray')
plt.title('%i' % train_data.train_labels[2])
plt.show()
输出的图片是 4
```




```
    train_loader = Data.DataLoader(dataset=train_data,
batch_size=BATCH_SIZE, shuffle=True)
    #加载数据进行训练，每次加载 50 张图片，大小为 28×28


    class AutoEncoder(nn.Module):
        def __init__(self):
            super(AutoEncoder, self).__init__()

            self.encoder = nn.Sequential(
                nn.Linear(28*28, 128),
                nn.Tanh(),
                nn.Linear(128, 64),
                nn.Tanh(),
                nn.Linear(64, 12),
                nn.Tanh(),
                nn.Linear(12, 3),   # compress to 3 features which can
be visualized in plt
            )
```

输入的图片大小为 28×28 像素大小，经过第一层的隐藏层，128 个神经元经过 Tanh 函数线性变换，然后再经过第二层的隐藏层，由 128 个神经元变成 64 个神经元，经过 Tanh 激活函数，进行线性变换，然后再经过第三层的隐藏层，由 64 个神经元变成 12 个神经元，经过 Tanh 激活函数，进行线性变换，然后再经过第四层的隐藏层，由 12 个神经元变成 3 个神经元，进行线性变换。上述步骤，我们完成了图片压缩步骤。下面我们来看看如

何进行解压操作。

```
            self.decoder = nn.Sequential(
                nn.Linear(3, 12),
                nn.Tanh(),
                nn.Linear(12, 64),
                nn.Tanh(),
                nn.Linear(64, 128),
                nn.Tanh(),
                nn.Linear(128, 28*28),
                nn.Sigmoid(),       # compress to a range (0, 1)
            )
```

输入的图片经过压缩，从 128 个神经元压缩到 3 个神经元，现在，如何把压缩后的图片像素转换成原始图片的像素呢？首先，解压操作经过第一层的隐藏层，3 个神经元经过 Tanh 函数线性变换，由 3 个神经元变成 12 个神经元，然后再经过第二层的隐藏层，由 12 个神经元变成 64 个神经元，经过 Tanh 激活函数，进行线性变换，然后再经过第三层的隐藏层，由 64 个神经元变成 128 个神经元，经过 Tanh 激活函数，进行线性变换，然后再经过第四层的隐藏层，由 128 个神经元变成 784 个神经元，进行线性变换，得到 784 个像素值。上述步骤，我们完成了图片解压步骤。

```
        def forward(self, x):
            encoded = self.encoder(x)
            decoded = self.decoder(encoded)
            return encoded, decoded
    autoencoder = AutoEncoder()
    optimizer = torch.optim.Adam(autoencoder.parameters(), lr=LR)
    class torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999),
eps=1e-08, weight_decay=0)
```

参数说明如下。

params（iterable）：待优化参数的 iterable 或者是定义了参数组的 dict。

lr（float，可选）：学习速率（默认：1e-3）。

betas（Tuple[float，float]，可选）：用于计算梯度以及梯度平方的运行平均值的系数（默认：0.9，0.999）。

eps（float，可选）：为了增加数值计算的稳定性而加到分母里的项（默认：1e-8）。

weight_decay（float，可选）：权重衰减（L2 惩罚）（默认：0）。

```
loss_func = nn.MSELoss()

class torch.nn.MSELoss(size_average=True)
#创建一个衡量输入 x(模型预测输出)和目标 y 之间均方误差标准。
loss(x,y)=1/n∑(xi-yi)2
```

x 和 y 可以是任意形状，每个包含 *n* 个元素。

对 *n* 个元素对应的差值的绝对值求和，得出来的结果除以 *n*。

如果在创建 MSELoss 实例的时候，在构造函数中传入 size_average=False，那么求出来的平方和将不会除以 *n*。

```
f, a = plt.subplots(2, N_TEST_IMG, figsize=(5, 2))
plt.ion()
#画图，子图个数为 5×2
view_data=Variable(train_data.train_data[:N_TEST_IMG].view(-1,
28*28).type(torch.FloatTensor)/255.)
#输入的图片大小经过变换
for i in range(N_TEST_IMG):

a[0][i].imshow(np.reshape(view_data.data.numpy()[i],(28,28)),cmap
='gray'); a[0][i].set_xticks(()); a[0][i].set_yticks(())
#显示前五张图片
for epoch in range(EPOCH):
    for step, (x, y) in enumerate(train_loader):
        b_x = Variable(x.view(-1, 28*28))   # batch x, shape (batch,
28*28)
        b_y = Variable(x.view(-1, 28*28))   # batch y, shape (batch,
28*28)
        b_label = Variable(y)
        encoded, decoded = autoencoder(b_x)
        loss = loss_func(decoded, b_y)
#计算损失函数
        optimizer.zero_grad()
```

```
    #把梯度设置为零
            loss.backward()
    #反向传播，计算梯度
            optimizer.step()

            if step % 100 == 0:
                print('Epoch: ', epoch, '| train loss: %.4f' %
loss.data[0])
    #每 100 次训练输出误差
                _, decoded_data = autoencoder(view_data)
                for i in range(N_TEST_IMG):
                    a[1][i].clear()

a[1][i].imshow(np.reshape(decoded_data.data.numpy()[i], (28, 28)),
cmap='gray')
                    a[1][i].set_xticks(()); a[1][i].set_yticks(())
                plt.draw(); plt.pause(0.05)

    plt.ioff()
    plt.show()
    #输出剩下的 5 张图片
```

```
    #输出误解结果：
    Epoch:  0 | train loss: 0.0691
    Epoch:  0 | train loss: 0.0601
    Epoch:  0 | train loss: 0.0597
    Epoch:  0 | train loss: 0.0575
    Epoch:  9 | train loss: 0.0314
    Epoch:  9 | train loss: 0.0342
    Epoch:  9 | train loss: 0.0336
    Epoch:  9 | train loss: 0.0350
    Epoch:  9 | train loss: 0.0351
```

经过训练之后，误差为 0.0351，相对来说还是比较准确的。

第 10 章

10

对抗生成网络

Ian J. Goodfellow 在 2014 年的 *Generative Adversative Nets* 论文中，第一次提出了对抗网络模型，如今对抗网络模型在深度学习生成模型领域已经取得了不错的成果。论文提出利用对抗过程估计生成模型，可以认为是在无监督表示学习（Unsuperivised representation learning）上的一个突破，现在主要的应用是用其生成自然图片（natural images）。包括 Goodfellow 如今所在的 OpenAI 公司一直在致力于研究推广 GAN，并将其应用在不同的任务上。同时 Facebook 和 Twitter 最近两年也投入了大量的精力来研究，并将 GAN 应用在了图像生成和视频生成上。

GAN 的原理很简单，简单说是概率生成模型的目的，就是找出给定观测数据内部的统计规律，并且能够基于所得到的概率分布模型，产生全新的，与观测数据类似的数据。下面是 Ian J. Goodfellow 在论文中对 GAN 的简单介绍，如图 10.1 所示。

The adversarial modeling framework is most straightforward to apply when the models are both multilayer perceptrons. To learn the generator's distribution p_g over data $\boldsymbol{x}$, we define a prior on input noise variables $p_{\boldsymbol{z}}(\boldsymbol{z})$, then represent a mapping to data space as $G(\boldsymbol{z};\theta_g)$, where G is a differentiable function represented by a multilayer perceptron with parameters θ_g. We also define a second multilayer perceptron $D(\boldsymbol{x};\theta_d)$ that outputs a single scalar. $D(\boldsymbol{x})$ represents the probability that $\boldsymbol{x}$ came from the data rather than p_g. We train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize $\log(1-D(G(\boldsymbol{z})))$. In other words, D and G play the following two-player minimax game with value function $V(G,D)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1-D(G(\boldsymbol{z})))]. \quad (1)$$

图 10.1　*Generative Adversative Nets* 论文对 GAN 的简单介绍

由于我们需要大量的先验知识去对真实世界进行建模，其中包括选择什么样的先验知识、什么样的分布等。而建模的好坏直接影响着我们的生成模型的表现。

真实世界的数据往往很复杂，我们要用来拟合模型的计算量往往非常庞大，甚至难以承受。

GAN 很好地解决了这两大难题。GAN 网络的巧妙在于其设计思维，而技术上是对现有算法的组合，GAN 网络主要由两个网络合成。一个是 G 生成网络：输入为随机数，输出为生成数据。目的是为了生成数据的取值范围与真实数据相似，具体使用什么函数视情况而定。另一个是 D 区分网络。D 区分网络输入数据为混合 G 的输出数据及样本数据。输出一个判别概率。Ian J. Goodfellow 在论文中指出训练方式 G 网络的 loss 是 log（1-D(G(z))，而 D 的 loss 是-(log(D(x)) + log(1-D(G(z)))。

为了使 G 生成网络的损失函数 loss 最小，在 G 生成网络的训练的时候希望 log(1-D(G(z))最小，而 D 的 loss 损失函数是-(log(D(x)) + log(1-D(G(z))) 希望真实数据的 D 区分网络输出趋近于 1，而生成数据的输出即 D(G(z))趋近于 0。从而分清楚真实数据和生成数据。由于 GAN 是一个非常灵活的设计框架，各种类型的损失函数都可以整合到 GAN 模型当中，这样针对不同的任务，我们可以设计不同类型的损失函数，都会在 GAN 的框架下进行学习和优化。

在 Ian J. Goodfellow 的 *Generative Adversative Nets* 论文中有下面一幅图片，如图 10.2 所示，图中点线为真实数据分布，实线为生成的数据分布，虚线表示生成的数据对应于真实数据分布的概率。（a）中是初始状态，由于刚开始训练，需要更新参数，经过若干次训练之后，实线能够趋近于圆点线，直到更新 G 的模型使其生成的数据分布更加趋近与真实数据分布。从（a）图训练到（b）图，可以很明显地看到随着实线向着圆点线的偏移，实线开始逐渐地下降。随后经过逐步训练如（c），实线逐渐向圆点线的线靠拢，直到训练形成（d）为止。此时，G 网络和 D 网络就处于平衡状态，无法再进一步更新了。

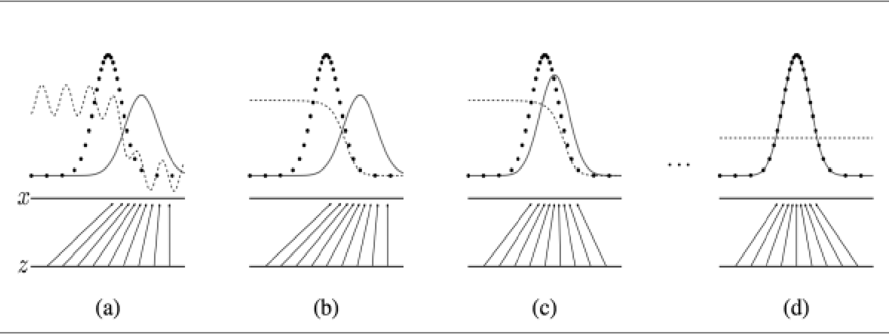


图 10.2　GAN 的训练过程

图 10.3 中的 Z 是 G 的输入，一般情况下是高斯随机分布生成的数据。其中 G 的输出是 G(z)，对于真实的数据，一般都为图片，将分布变量用 X 来表示。那么对于 D 的输出判断则是来自 X 的可能性，是一个常量。

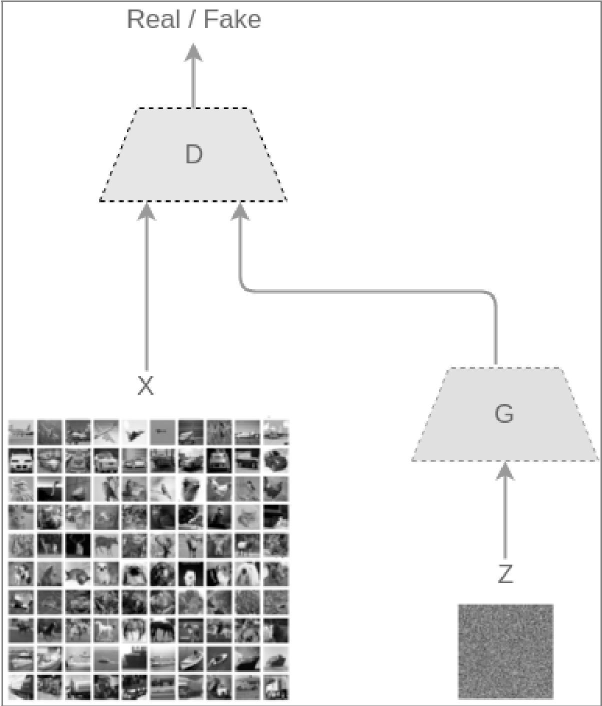


图 10.3　GAN 的流程图

由于生成对抗网络的优点不需要大量的带标签的数据，损失函数来源

于 D 上午判定，同时产生大量生成数据用于训练，相当于接近无监督学习，在一定的情况下可以和深度神经网络结合，比如说与卷积神经网络结合的 DCGAN。由于产生大量生成数据没有推导过程，缺少数学理论，生成器，判别器需要共同训练，导致训练难度加大，容易出现训练失败。

GAN 最直接的应用，就是用于真实数据分布的建模和生成，包括可以生成一些图像和视频，以及生成一些自然语句和音乐等。由于 GAN 可以生成大量数据，可以解决一些传统的机器学习中所面临的数据不足的问题，因此可以应用在半监督学习、无监督学习、多视角、多任务学习的任务中。

10.1　DCGAN原理

在 GAN 的基础上，DCGAN 的一大特点就是使用了卷积层。DCGAN 的基本架构就是使用几层“反卷积”（Deconvolution）网络，如图 10.4 所示。“反卷积”类似于一种反向卷积，这跟用反向传播算法训练监督的卷积神经网络（CNN）是类似的操作。DCGAN 对卷积神经网络的结构做了一些改变，以提高样本的质量和收敛的速度，这些改变如下。

取消所有 pooling 层。G 网络中使用反卷积（Deconvolutional layer）进行上采样，D 网络中用加入 stride 的卷积代替 pooling。

D 和 G 中均使用 batch normalization。

去掉 FC 层，使网络变为全卷积网络。

G 网络中使用 ReLU 作为激活函数，最后一层使用 tanh。

D 网络中使用 LeakyReLU 作为激活函数，最后一层使用 softmax。

Generative networks 的架构。

在模型的细节处理上，预处理环节，将图像 scale 到 tanh 的[-1，1]。

mini-batch 训练，batch size 是 128。

所有的参数初始化在（0，0.02）的正态分布中随即得到。

LeakyReLU 的斜率是 0.2。

虽然之前的 GAN 使用 momentum 来加速训练，DCGAN 使用调好超参的 Adam optimizer。

learning rate=0.0002。

将 momentum 参数 beta 从 0.9 降为 0.5 来防止震荡和不稳定。

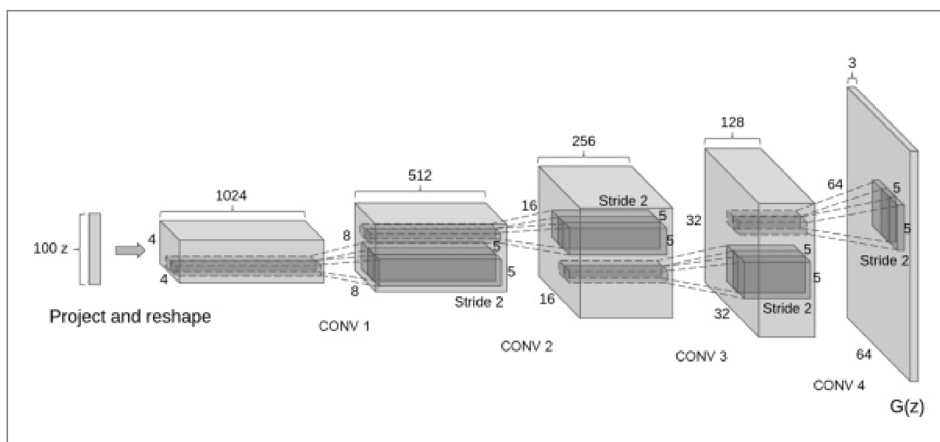


图 10.4　DCGAN 模型

图 10.5 等式左边都是噪声 z（一般为均匀噪声）经过 G(z)产生的人脸图片。

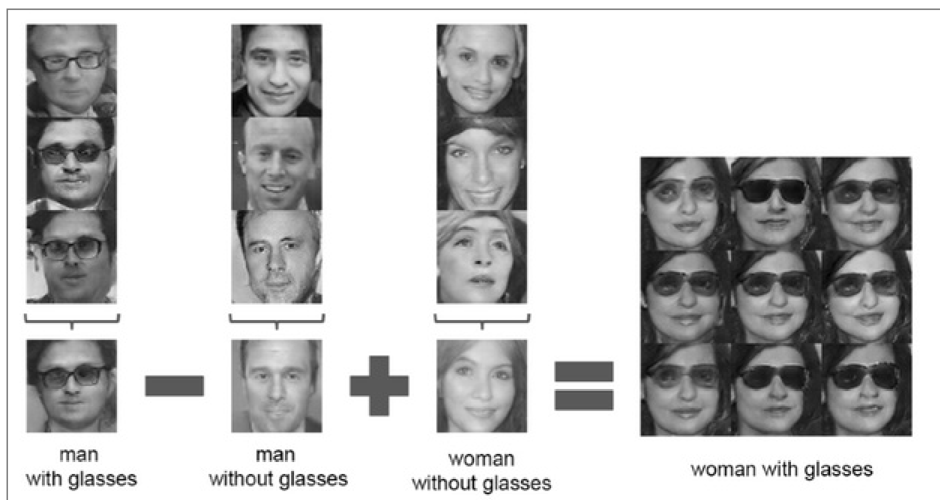


图 10.5　DCGAN 模型案例

使用 GAN 作为特征提取器分类 CIFAR-10，如图 10.6 所示。虽然 2015 年 Dosovitskiy 等提出 DCGAN 的性能仍然比不上典型的 CNN，但仍然取得较好的效果。

Model	Accuracy	Accuracy(400 per class)	Max # of features units
1 Layer k-means	80.6%	63.7%(±0.7%)	4800
3 Layer K-means Learned RF	82.0%	70.7%(±0.7%)	3200
View Invariant K-means	81.9%	72.6%(±0.7%)	6400
Exemplar CNN	84.3%	77.4%(±0.2%)	1024
DCGAN(ours)+L2-SVM	82.8%	73.8%(±0.4%)	512

图 10.6　GAN 作为特征提取器分类 CIFAR-10 结果

在 GAN 的基础上，经过简单的改造，把纯无监督的 GAN 变成半监督或者有监督的，为 GAN 的训练加上束缚。例如 Conditional Generative Adversarial Nets（CGAN）模型。在生成模型（G）和判别模型（D）的建模中加入标签。因此，CGAN 可以看做把无监督的 GAN 变成有监督的模型。

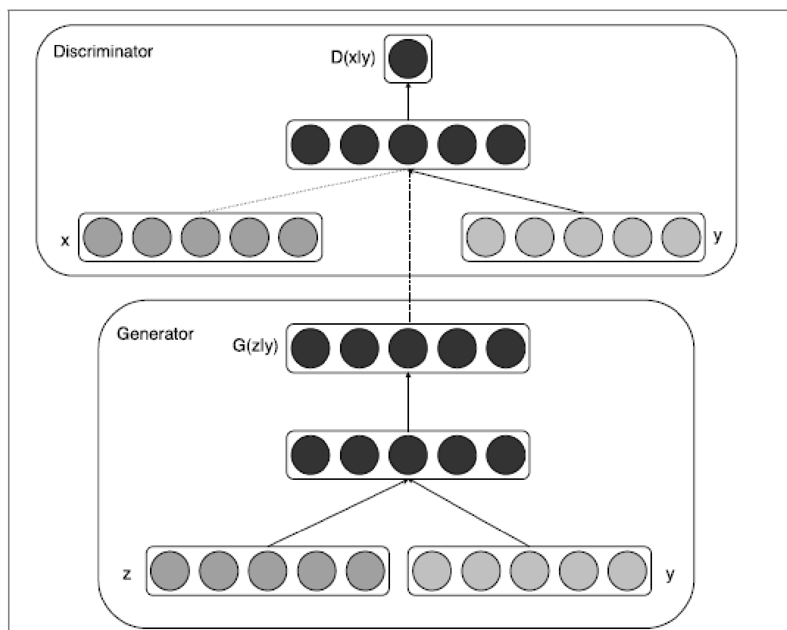


图 10.7　CGAN 模型

就是 G 网络的输入在 z 的基础上连接一个输入 y，然后在 D 网络的输入在 x 的基础上也连接一个 y。

从流程图 10.7 可以看出训练方式几乎就是不变的，但是 GAN 从无监督变成了有监督。

我们来看看 CGAN 的应用，如图 10.8 所示，利用 CGAN 进行文字和位置约束来生成图片。在图片的特定位置约束，同时加上相应的标签文字，作为随机输入，生成图片，然后与真实图片做对比，进行判断图片真假。

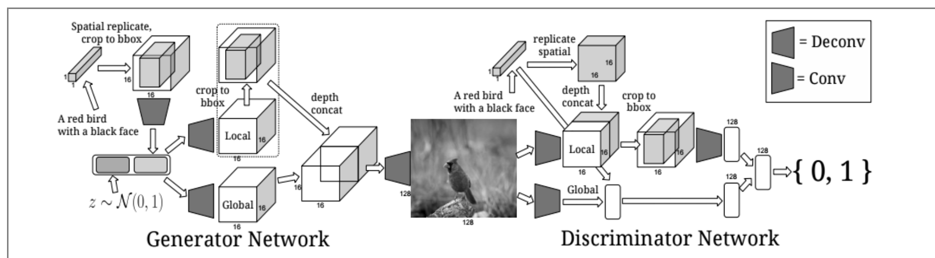


图 10.8　利用 CGAN 进行文字和位置约束来生成图片

图 10.9 是效果图。

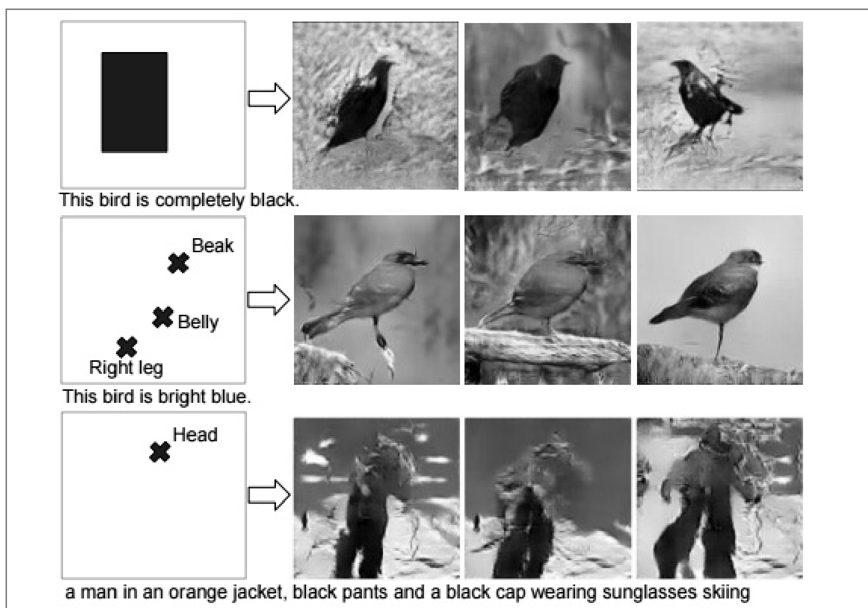


图 10.9　利用 CGAN 进行文字和位置约束来生成图片效果图

图 10.10 是利用 CGAN 生成关键点多层次参与图片生成模型流程。

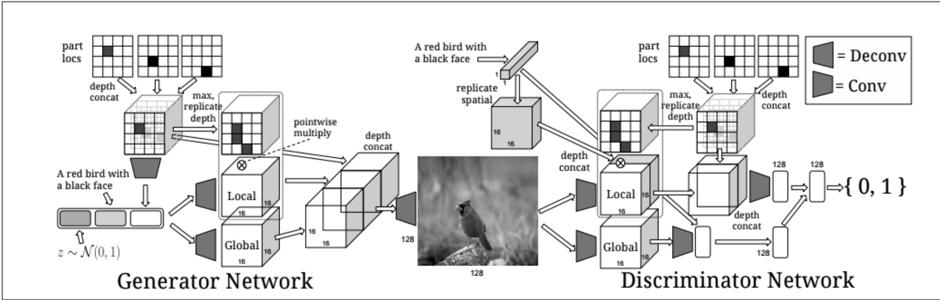


图 10.10　CGAN 生成关键点多层次参与图片生成模型

比如说我们要在图片的具体位置进行图片的伸缩、平移、扩张等。如图 10.11 所示是对框架里面的鸟儿进行缩小、平移、伸缩的案例。

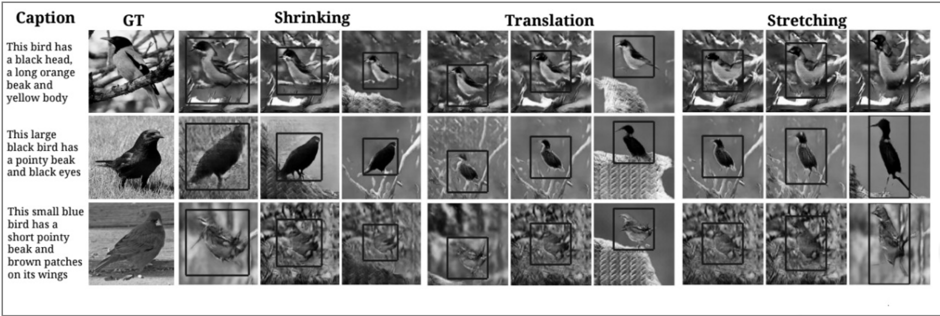


图 10.11　CGAN 生成关键点多层次参与图片生成模型实例

Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network 这篇文章将对抗学习用于基于单幅图像的高分辨重建。

无论是 GAN 还是 DCGAN，这种对抗学习的方式，是一种比较成功的生成模型，可以从训练数据中学习到近似的分布情况，那么有了这个分布，自然可以应用到很多领域。比如图像的修复，图像的超分辨率，图像翻译等。

在 GAN 基础上还有 NIPS2016 的 InfoGAN 模型。InfoGAN 的目标就是通过非监督学习得到可分解的特征表示。使用 GAN 加上最大化生成的图片和输入编码之间的互信息。最大的好处就是可以不需要监督学习，而且不

需要大量额外的计算就能得到可解释的特征。

自从 2014 年 Ian Goodfellow 提出 GAN 以来，GAN 就存在着训练困难、生成器和判别器的 loss 无法指示训练进程、生成样本缺乏多样性等问题。Wasserstein GAN（下面简称 WGAN）彻底解决了 GAN 训练不稳定的问题，不再需要小心平衡生成器和判别器的训练程度。同时，训练过程中有一个数值指示训练的进程，这个数值越小代表 GAN 训练得越好，代表生成器产生的图像质量越高，只需要最简单的多层全连接网络就可以做到网络结构设计。由于这些网络比较复杂，有兴趣的读者可以自行查阅学习。

10.2 GAN对抗生成网络实例

在 PyTorch 中实现对抗生成网络。

本例文件名为 PyTorch/Chapter11/pt1_generative_adversarial_network.py。

```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets
from torchvision import transforms
from torchvision.utils import save_image
from torch.autograd import Variable

#导入模块
def to_var(x):
    if torch.cuda.is_available():
        x = x.cuda()
    return Variable(x)
#利用 CUDA 对数据处理加速
def denorm(x):
    out = (x + 1) / 2
    return out.clamp(0, 1)
```

```
transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=(0.5, 0.5, 0.5),
                                 std=(0.5, 0.5, 0.5))])
#对图像进行处理，将多个transform组合起来使用

mnist = datasets.MNIST(root='./data/',
                   train=True,
                   transform=transform,
                   download=True)
# 下载MNIST图像数据
data_loader = torch.utils.data.DataLoader(dataset=mnist,
                                   batch_size=100,
                                   shuffle=True)
# 加载数据
D = nn.Sequential(
   nn.Linear(784, 256),
   nn.LeakyReLU(0.2),
   nn.Linear(256, 128),
   nn.LeakyReLU(0.2),
   nn.Linear(128, 1),
   nn.Sigmoid())
```

D 区分网络输入的数据经过线性函数，对输入数据做线性变换：$y=Ax+b$。

```
G = nn.Sequential(
   nn.Linear(64, 128),
   nn.LeakyReLU(0.2),
   nn.Linear(128, 256),
   nn.LeakyReLU(0.2),
   nn.Linear(256, 784),
   nn.Tanh())
```

G 生成网络目的是为了生成数据的取值范围与真实数据相似。

```
if torch.cuda.is_available():
   D.cuda()
```

```
G.cuda()
#利用 cuda 加速
criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0003)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0003)
#损失函数以及优化函数
for epoch in range(200):
    for i, (images, _) in enumerate(data_loader):
        batch_size = images.size(0)
        images = to_var(images.view(batch_size, -1))
        real_labels = to_var(torch.ones(batch_size))
        fake_labels = to_var(torch.zeros(batch_size))
        outputs = D(images)
        d_loss_real = criterion(outputs, real_labels)
        real_score = outputs
        z = to_var(torch.randn(batch_size, 64))
        fake_images = G(z)
        outputs = D(fake_images)
        d_loss_fake = criterion(outputs, fake_labels)
        fake_score = outputs
        d_loss = d_loss_real + d_loss_fake
        D.zero_grad()
        d_loss.backward()
        d_optimizer.step()
        z = to_var(torch.randn(batch_size, 64))
        fake_images = G(z)
        outputs = D(fake_images)
        g_loss = criterion(outputs, real_labels)
        D.zero_grad()
        G.zero_grad()
        g_loss.backward()
        g_optimizer.step()
```

GAN 启发自博弈论中的二人零和博弈（Two-player Game），GAN 模型中的两位博弈方分别由生成式模型（Generative Model）和判别式模型（Discriminative Model）充当。生成模型 G 捕捉样本数据的分布，用服从某

一分布（均匀分布，高斯分布等）的噪声 z 生成一个类似真实训练数据的样本，追求效果越像真实样本越好；判别模型 D 是一个二分类器，估计一个样本来自训练数据（而非生成数据）的概率，如果样本来自于真实的训练数据，D 输出大概率，否则，D 输出小概率。可以做如下类比：生成网络 G 好比假币制造团伙，专门制造假币，判别网络 D 好比警察，专门检测使用的货币是真币还是假币，G 的目标是想方设法生成和真币一样的货币，使得 D 判别不出来，D 的目标是想方设法检测出 G 生成的是假币。

训练的过程中固定的一方更新另一方的网络权重，交替迭代，在这个过程中，双方都极力优化自己的网络，从而形成竞争对抗，直到双方达到一个动态的平衡（纳什均衡），此时生成模型 G 恢复了训练数据的分布（造出了和真实数据一模一样的样本），判别模型再也判别不出来结果，准确率为 50%，约等于乱猜。

当固定生成网络 G 的时候，对于判别网络 D 的优化，可以这样理解：输入来自真实数据，D 优化网络结构使自己输出 1，输入来自生成数据，D 优化网络结构使自己输出 0；当固定判别网络 D 的时候，G 优化自己的网络使自己输出尽可能和真实数据一样的样本，并且使得生成的样本经过 D 的判别之后，D 输出高概率。

```
    #对数据进行训练
            if (i+1) % 300 == 0:
                print('Epoch [%d/%d], Step[%d/%d], d_loss: %.4f, '
                      'g_loss: %.4f, D(x): %.2f, D(G(z)): %.2f'
                      %(epoch, 200, i+1, 600, d_loss.data[0],
g_loss.data[0],
                        real_score.data.mean(),
fake_score.data.mean()))
    #对图片进行训练
        if (epoch+1) == 1:
            images = images.view(images.size(0), 1, 28, 28)
            save_image(denorm(images.data),
'./data/real_images.png')
        fake_images = fake_images.view(fake_images.size(0), 1, 28,
28)
```

```
        save_image(denorm(fake_images.data),
'./data/fake_images-%d.png' %(epoch+1))
    torch.save(G.state_dict(), './generator.pkl')
    torch.save(D.state_dict(), './discriminator.pkl')
    #保存结果
```

从细节上来看，生成模型可以做一些无中生有的事情。比如图片的高清化，遮住图片的一部分去修复，再或者画了一幅人脸的肖像轮廓，将其渲染成栩栩如生的照片等。

再提高一层，生成模型的终极是创造，通过发现数据里的规律来生产一些东西，这就和真正的人工智能对应起来了。一个人，他可以通过看、听、闻去感知这世界，这是所谓的识别，他也可以说、画、想一些新的事情，这就是创造。所以，生成模型我认为是 AI 在识别任务发展相当成熟之后，AI 发展的又一个阶段。

风格迁移（Style Transfer）是深度学习众多应用中非常有趣的一种，如图 10.13 所示，我们可以使用这种方法把一张图片的风格“迁移”到另一张图片上



图 10.13　风格迁移实例

原始的风格迁移速度是非常慢的。在 GPU 上，生成一张图片都需要 10 分钟左右，而如果只使用 CPU 而不使用 GPU 运行程序，甚至需要几个小时。这个时间还会随着图片尺寸的增大而迅速增大。

这其中的原因在于，在原始的风格迁移过程中，把生成图片的过程当作一个“训练”的过程。每生成一张图片，都相当于要训练一次模型，这中间可能会迭代几百几千次。图像风格转移，直观来看，就是将一幅图片的“风格”转移到另一幅图片，而保持它的内容不变。一般我们将内容保持不变的图称为内容图，content image，把含有我们想要的风格的图片，如

梵高的星空，称为风格图，style image。

其实要实现的东西很清晰，就是需要将两张图片融合在一起，这个时候就需要定义怎么才算融合在一起。首先需要的就是内容上是相近的，然后风格上是相似的。这样我们就知道需要做的事情是什么了，我们需要计算融合图片和内容图片的相似度，或者说差异性，然后尽可能降低这个差异性；同时我们也需要计算融合图片和风格图片在风格上的差异性，然后也降低这个差异性就可以了。这样我们就能够量化我们的目标了。

利用深度学习模型，我们可以做许多有趣的事情。

11

第 11 章

Seq2seq 自然语言处理

11.1 Seq2seq自然语言处理简介

使用计算机对自然语言进行处理，便需要将自然语言处理成机器能够识别的符号，在机器学习过程中，就需要将其进行数值化。而词是自然语言理解与处理的基础，因此需要对词进行数值化，词向量便是一种可行又有效的方法。何为词向量，即使用一个指定长度的实数向量 V 来表示一个词。有一种最简单的表示方法，就是使用 One-hot vector 表示单词，即根据单词的数量$|V|$生成一个$|V|\times 1$ 的向量，当某一位为 1 的时候其他位都为 0，然后这个向量就代表一个单词。

目前比较流行的 Seq2Seq 模型，由 Sutskever 等人提出，基于一个 Encoder-Decoder 的结构，将 Source 句子先 Encode 成一个固定维度 d 的向量，然后通过 Decoder 部分一个字符一个字符生成 Target 句子。加入了 Attention 注意力分配机制后，使得 Decoder 在生成新的 Target Sequence 时，能得到之前 Encoder 编码阶段每个字符的隐藏层的信息向量 Hidden State，使得生成新序列的准确度提高。Seq2seq 模型就像一个翻译模型，输入是一个序列（比如一个英文句子），输出也是一个序列（比如该英文句子所对应的法文翻译）。这种结构最重要的地方在于输入序列和输出序列的长度是可变的。

比如在机器翻译中输入（hello）-> 输出（你好）。输入是 1 个英文单词，输出为两个汉字。

在对话机器中我们提（输入）一个问题，机器会自动生成（输出）回答。这里的输入和输出显然是长度没有确定的序列（Sequences）。

最基础的 Seq2Seq 模型包含了三个部分，即 Encoder、Decoder 以及连接两者的中间状态向量，Encoder 通过学习输入，将其编码成一个固定大小的状态向量 S，继而将 S 传给 Decoder，Decoder 再通过对状态向量 S 的学习来进行输出。下面是它的工作原理。

有一个 RNN 层（或其堆叠）作为“编码器”：它负责处理输入序列并返回其自身的内部状态。注意，我们将丢弃编码器 RNN 的输出，只恢复状态。该状态将在下一步骤中用作解码器的“上下文”或“环境”。

另外还有一个 RNN 层（或其堆叠）作为“解码器”：在给定目标序列前一个字符的情况下，对其进行训练以预测目标序列的下一个字符。具体来说，就是训练该层，使其能够将目标序列转换成将来偏移了一个时间步长的同一个序列，这种训练过程被称为“Teacher Forcing（老师强迫）”。有一点很重要，解码器将来自编码器的状态向量作为初始状态，这样，解码器就知道了它应该产生什么样的信息。实际上就是解码器以输入序列为条件，对于给定的 targets[...t]学习生成 targets[t+1...]。

其实基础的 Seq2Seq 是有很多弊端的，首先 Encoder 将输入编码为固定大小状态向量的过程，实际上是一个“信息有损压缩”的过程，如果信息量越大，那么这个转化向量的过程对信息的损失就越大，同时，随着 sequence length 的增加，意味着时间维度上的序列很长，RNN 模型也会出现梯度弥散。最后，基础的模型连接 Encoder 和 Decoder 模块的组件仅仅是一个固定大小的状态向量，这使得 Decoder 无法直接去关注输入信息的更多细节。由于基础 Seq2Seq 的种种缺陷，之后引入了 Attention 的概念以及 Bi-directional encoder layer 等。

Seq2seq 其实可以用在很多地方，比如机器翻译、自动对话机器人、文档摘要自动生成、图片描述自动生成。比如 Google 就基于 Seq2seq 开发了一个对话模型，使用两个 LSTM 的结构，LSTM1 将输入的对话编码成一个

固定长度的实数向量，LSTM2 根据这个向量不停地预测后面的输出（解码）。只是在对话模型中，使用的语料是（你说的话-我答的话）这种类型的 pairs 。而在机器翻译中使用的语料是（hello-你好）这样的 pairs。

此外，如果我们的输入是图片，输出是对图片的描述，用这样的方式来训练的话就能够完成图片描述的任务。

可以看出来，Seq2seq 具有非常广泛的应用场景，而且效果也是非常明显的。同时，因为是端到端的模型（大部分的深度模型都是端到端的），它减少了很多人工处理和规则制定的步骤。在 Encoder-Decoder 的基础上，人们又引入了 Attention Mechanism 等技术，使得这些深度方法在各个任务上表现更加突出。

11.2　Seq2seq自然语言处理案例

接下来我们用 PyTorch 来实现 Seq2seq 自然语言处理。

本例文件名为 PyTorch/Chapter12/pt2_seq2seq_translation_tutorial.py。

我们接下来通过神经网络来实现用法语翻译英语。

```
    from __future__ import unicode_literals, print_function,
division
    from io import open
    import unicodedata
    import string
    import re
    import random

    import torch
    import torch.nn as nn
    from torch.autograd import Variable
    from torch import optim
    import torch.nn.functional as F
    #导入模块包

    use_cuda = torch.cuda.is_available()
```

```
#利用 CUDA 进行加速训练
SOS_token = 0
EOS_token = 1
class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2
#SOS 表示一句话的开始标准，EOS 表示一句话的结束标志。
    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)
#对传进来的句子进行分割
    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
#对一句话中出现的没有的词建立新的一一对应关系。
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )
def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
#我们将 Unicode 字符的 ASCII，利用正则表达式进行筛选，让所有字符小写，删除大部分标点符号。
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")
```

```
        lines = open('data/%s-%s.txt' % (lang1, lang2),
encoding='utf-8').\
            read().strip().split('\n')

        pairs = [[normalizeString(s) for s in l.split('\t')] for l
in lines]

        if reverse:
            pairs = [list(reversed(p)) for p in pairs]
            input_lang = Lang(lang2)
            output_lang = Lang(lang1)
        else:
            input_lang = Lang(lang1)
            output_lang = Lang(lang2)

        return input_lang, output_lang, pairs
    #读取文件，并且对文件中的文本进行分割
    MAX_LENGTH = 10

    eng_prefixes = (
        "i am ", "i m ",
        "he is", "he s ",
        "she is", "she s",
        "you are", "you re ",
        "we are", "we re ",
        "they are", "they re "
    )
    def filterPair(p):
        return len(p[0].split(' ')) < MAX_LENGTH and \
            len(p[1].split(' ')) < MAX_LENGTH and \
            p[1].startswith(eng_prefixes)



    def filterPairs(pairs):
        return [pair for pair in pairs if filterPair(pair)]
    #为了加快速度，每句话最多选取 10 个词。为了加快训练速度，删掉相对较短的句
子。
```

```
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2,
reverse)
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, pairs
input_lang, output_lang, pairs = prepareData('eng', 'fra', True)
print(random.choice(pairs))
#加载需要进行翻译的语句
#输出
Reading lines...
Read 135842 sentence pairs
Trimmed to 10853 sentence pairs
Counting words...
Counted words:
fra 4489
eng 2925
['vous etes arrogant .', 'you re arrogant .']

#定义输入模型
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, n_layers=1):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
```

```
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        for i in range(self.n_layers):
            output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
            return result.cuda()
        else:
            return result
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, n_layers=1):
        super(DecoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        for i in range(self.n_layers):
            output = F.relu(output)
            output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
            return result.cuda()
        else:
            return result
```

这部分为解码器，就是对上述的语句经过词向量之后进行解码翻译成英语。

```
    class AttnDecoderRNN(nn.Module):
        def __init__(self, hidden_size, output_size, n_layers=1,
dropout_p=0.1, max_length=MAX_LENGTH):
            super(AttnDecoderRNN, self).__init__()
            self.hidden_size = hidden_size
            self.output_size = output_size
            self.n_layers = n_layers
            self.dropout_p = dropout_p
            self.max_length = max_length

            self.embedding = nn.Embedding(self.output_size,
self.hidden_size)
            self.attn = nn.Linear(self.hidden_size * 2,
self.max_length)
            self.attn_combine = nn.Linear(self.hidden_size * 2,
self.hidden_size)
            self.dropout = nn.Dropout(self.dropout_p)
            self.gru = nn.GRU(self.hidden_size, self.hidden_size)
            self.out = nn.Linear(self.hidden_size, self.output_size)

        def forward(self, input, hidden, encoder_outputs):
            embedded = self.embedding(input).view(1, 1, -1)
            embedded = self.dropout(embedded)

            attn_weights = F.softmax(
                self.attn(torch.cat((embedded[0], hidden[0]), 1)),
dim=1)
            attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                     encoder_outputs.unsqueeze(0))

            output = torch.cat((embedded[0], attn_applied[0]), 1)
            output = self.attn_combine(output).unsqueeze(0)

            for i in range(self.n_layers):
                output = F.relu(output)
                output, hidden = self.gru(output, hidden)

            output = F.log_softmax(self.out(output[0]), dim=1)
```

```
        return output, hidden, attn_weights

    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
            return result.cuda()
        else:
            return result
```

Attention 机制把源句子中对生成句子重要的关键词的权重进行提高，可以更准确地应答。Attention 机制应用在聊天机器人，机器翻译等领域，大大提高了翻译效果。

采用注意力集中机制能够让网络在解码的时候着重输出某些重要的部分。如图 11.1 表示编码输入到解码输出。

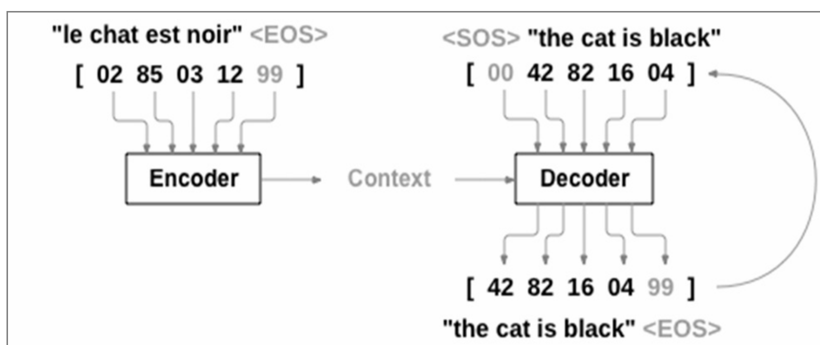


图 11.1 注意力集中机制

```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split('
')]
def variableFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    result = Variable(torch.LongTensor(indexes).view(-1, 1))
    if use_cuda:
        return result.cuda()
    else:
        return result
```

```
    def variablesFromPair(pair):
        input_variable = variableFromSentence(input_lang, pair[0])
        target_variable = variableFromSentence(output_lang,
pair[1])
        return (input_variable, target_variable)
```

我们需要一个输入张量（输入句子中词的索引）和目标张量（目标句子中词的索引）。在创建这些向量时我们将添加这两个序列的 EOS。

```
    teacher_forcing_ratio = 0.5
    def train(input_variable, target_variable, encoder, decoder,
encoder_optimizer, decoder_optimizer, criterion,
max_length=MAX_LENGTH):
        encoder_hidden = encoder.initHidden()
        encoder_optimizer.zero_grad()
        decoder_optimizer.zero_grad()
        input_length = input_variable.size()[0]
        target_length = target_variable.size()[0]
        encoder_outputs = Variable(torch.zeros(max_length,
encoder.hidden_size))
        encoder_outputs = encoder_outputs.cuda() if use_cuda else
encoder_outputs
        loss = 0
        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(
                input_variable[ei], encoder_hidden)
            encoder_outputs[ei] = encoder_output[0][0]
        decoder_input = Variable(torch.LongTensor([[SOS_token]]))
        decoder_input = decoder_input.cuda() if use_cuda else
decoder_input
        decoder_hidden = encoder_hidden
        use_teacher_forcing = True if random.random() <
teacher_forcing_ratio else False
        if use_teacher_forcing:
            for di in range(target_length):
                decoder_output, decoder_hidden, decoder_attention =
decoder(
                    decoder_input, decoder_hidden, encoder_outputs)
                loss += criterion(decoder_output,
```

```
target_variable[di])
                decoder_input = target_variable[di]  # Teacher
forcing
        else:
            for di in range(target_length):
                decoder_output, decoder_hidden, decoder_attention =
decoder(
                    decoder_input, decoder_hidden, encoder_outputs)
                topv, topi = decoder_output.data.topk(1)
                ni = topi[0][0]
                decoder_input = Variable(torch.LongTensor([[ni]]))
                decoder_input = decoder_input.cuda() if use_cuda else
decoder_input
                loss += criterion(decoder_output,
target_variable[di])
                if ni == EOS_token:
                    break
        loss.backward()
        encoder_optimizer.step()
        decoder_optimizer.step()
        return loss.data[0] / target_length
    #对模型进行训练
    def asMinutes(s):
        m = math.floor(s / 60)
        s -= m * 60
        return '%dm %ds' % (m, s)

    def timeSince(since, percent):
        now = time.time()
        s = now - since
        es = s / (percent)
        rs = es - s
        return '%s (- %s)' % (asMinutes(s), asMinutes(rs))
    #输出估计剩余时间和进度
    variable = training_pair[1]
            loss = train(input_variable, target_variable, encoder,
                         decoder, encoder_optimizer,
decoder_optimizer, criterion)
```

```
            print_loss_total += loss
            plot_loss_total += loss
            if iter % print_every == 0:
                print_loss_avg = print_loss_total / print_every
                print_loss_total = 0
                print('%s (%d %d%%) %.4f' % (timeSince(start, iter / 
n_iters),
                                             iter, iter / n_iters * 100, 
print_loss_avg))
            if iter % plot_every == 0:
                plot_loss_avg = plot_loss_total / plot_every
                plot_losses.append(plot_loss_avg)
                plot_loss_total = 0
        showPlot(plot_losses)
    #输出平均损失
    def showPlot(points):
        plt.figure()
        fig, ax = plt.subplots()
        # this locator puts ticks at regular intervals
        loc = ticker.MultipleLocator(base=0.2)
        ax.yaxis.set_major_locator(loc)
        plt.plot(point)

    #画图输出损失结果
    def evaluate(encoder, decoder, sentence, 
max_length=MAX_LENGTH):
        input_variable = variableFromSentence(input_lang, sentence)
        input_length = input_variable.size()[0]
        encoder_hidden = encoder.initHidden()
        encoder_outputs = Variable(torch.zeros(max_length, 
encoder.hidden_size))
        encoder_outputs = encoder_outputs.cuda() if use_cuda else 
encoder_outputs
        for ei in range(input_length):
            encoder_output, encoder_hidden = 
encoder(input_variable[ei],
                                                        encoder_hidden)
            encoder_outputs[ei] = encoder_outputs[ei] +
```

```
encoder_output[0][0]

        decoder_input = Variable(torch.LongTensor([[SOS_token]]))
# SOS
        decoder_input = decoder_input.cuda() if use_cuda else
decoder_input

        decoder_hidden = encoder_hidden

        decoded_words = []
        decoder_attentions = torch.zeros(max_length, max_length)

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention =
decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            ni = topi[0][0]
            if ni == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[ni])

            decoder_input = Variable(torch.LongTensor([[ni]]))
            decoder_input = decoder_input.cuda() if use_cuda else
decoder_input

        return decoded_words, decoder_attentions[:di + 1]
    def evaluateRandomly(encoder, decoder, n=10):
        for i in range(n):
            pair = random.choice(pairs)
            print('>', pair[0])
            print('=', pair[1])
            output_words, attentions = evaluate(encoder, decoder,
pair[0])
            output_sentence = ' '.join(output_words)
```

```
            print('<', output_sentence)
            print('')
    #进行评估并输出结果
    hidden_size = 256
    encoder1 = EncoderRNN(input_lang.n_words, hidden_size)
    attn_decoder1 = AttnDecoderRNN(hidden_size,
output_lang.n_words,
                                   1, dropout_p=0.1)
    if use_cuda:
        encoder1 = encoder1.cuda()
        attn_decoder1 = attn_decoder1.cuda()
    trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
    evaluateRandomly(encoder1, attn_decoder1)
    output_words, attentions = evaluate(
        encoder1, attn_decoder1, "je suis trop froid .")
    plt.matshow(attentions.numpy())
    def showAttention(input_sentence, output_words, attentions):
        fig = plt.figure()
        ax = fig.add_subplot(111)
        cax = ax.matshow(attentions.numpy(), cmap='bone')
        fig.colorbar(cax)
        ax.set_xticklabels([''] + input_sentence.split(' ') +
                        ['<EOS>'], rotation=90)
        ax.set_yticklabels([''] + output_words)
        ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
        ax.yaxis.set_major_locator(ticker.MultipleLocator(1))
        plt.show()
    def evaluateAndShowAttention(input_sentence):
        output_words, attentions = evaluate(
            encoder1, attn_decoder1, input_sentence)
        print('input =', input_sentence)
        print('output =', ' '.join(output_words))
        showAttention(input_sentence, output_words, attentions)
    evaluateAndShowAttention("elle a cinq ans de moins que moi .")
    evaluateAndShowAttention("elle est trop petit .")
    evaluateAndShowAttention("je ne crains pas de mourir .")
    evaluateAndShowAttention("c est un jeune directeur plein de
talent .")
```

```
#利用画图来输出效果
#输出结果
#输出剩余时间以及完成进度
Out:
1m 35s (- 22m 17s) (5000 6%) 2.8940
3m 7s (- 20m 21s) (10000 13%) 2.3590
4m 40s (- 18m 43s) (15000 20%) 2.0273
6m 14s (- 17m 10s) (20000 26%) 1.8096
7m 48s (- 15m 36s) (25000 33%) 1.5893
9m 21s (- 14m 2s) (30000 40%) 1.4345
10m 55s (- 12m 28s) (35000 46%) 1.2635
12m 28s (- 10m 55s) (40000 53%) 1.1544
14m 2s (- 9m 21s) (45000 60%) 1.0362
15m 35s (- 7m 47s) (50000 66%) 0.9439
17m 9s (- 6m 14s) (55000 73%) 0.8612
18m 44s (- 4m 41s) (60000 80%) 0.7582
20m 18s (- 3m 7s) (65000 86%) 0.7147
21m 51s (- 1m 33s) (70000 93%) 0.6659
23m 26s (- 0m 0s) (75000 100%) 0.6180
#输出损失函数数据
```

误差效果图如图 11.2 所示。

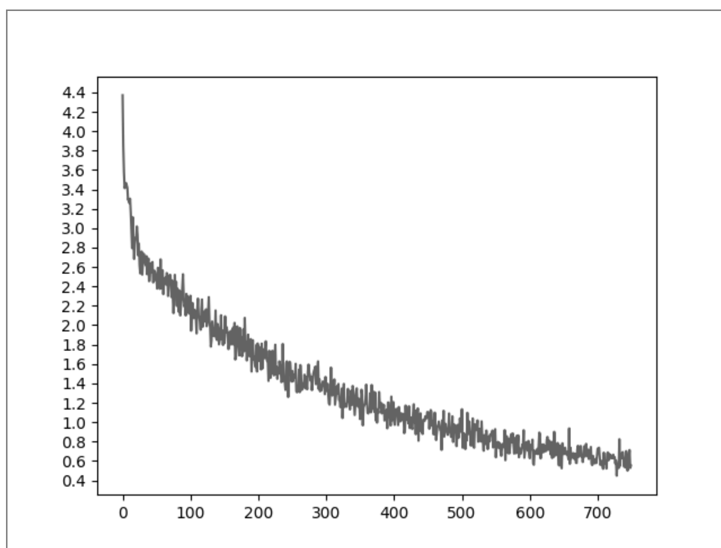


图 11.2 误差效果图

```
#输出经过训练之后翻译出来的句子
> c est une sorte de peintre .
= he is a sort of painter .
< he is a sort of a painter . <EOS>
> je suis pret a y aller maintenant .
= i m ready to go now .
< i m ready to go now . <EOS>
> nous sommes tous decus .
= we re all disappointed .
< we re all devastated . <EOS>
> il est accro a l hero .
= he s addicted to heroin .
< he is addicted to heroin . <EOS>
> je ne compare pas tom a mary .
= i m not comparing tom to mary .
< i m not comparing tom . . <EOS>
> il n est pas de tres bonne compagnie .
= he is not very good company .
< he is not very good company . <EOS>
> il est avec ses parents .
= he s with his parents .
< he is engaged with his parents . <EOS>
> je suis desolee de t avoir derange .
= i m sorry to have disturbed you .
< i m sorry to have disturbed you . <EOS>
> il manque de pratique .
= he s out of practice .
< he s out of practice . <EOS>
> j ai une faim de loup .
= i m as hungry as a bear .
< i m very hungry . <EOS>
```

其中“>”符号后面的句子表示待翻译的法文，“=”符号后面的句子是真是的英文翻译，“<”符号后面的句子是机器翻译后的结果。

注意力的权重可视化，如图 11.3 所示。

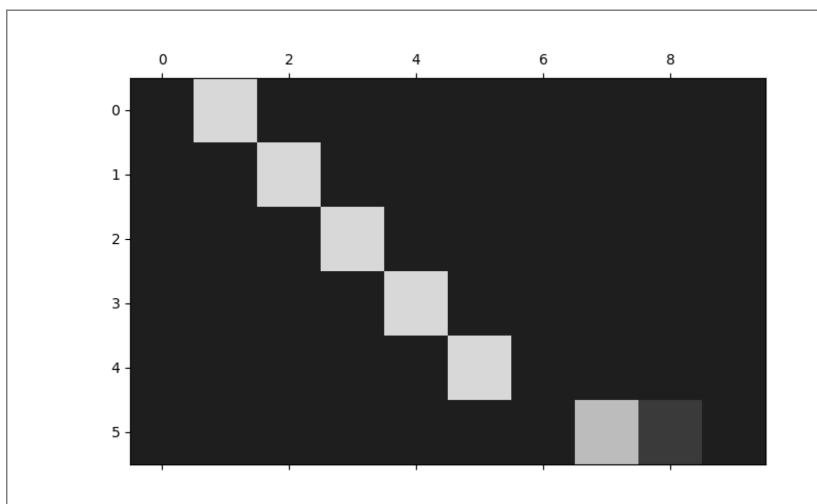


图 11.3　注意力的权重可视化

我们可以通过注意力权重可视化来观察每个单词的权重，来确定编码输出的重点部分，有利于我们理解网络的注意力在哪部分。

我们再来看一个注意力机制的例子。

```
input = elle a cinq ans de moins que moi .
input = elle est trop petit .
input = je ne crains pas de mourir .
input = c est un jeune directeur plein de talent .
```

图 11.4 所示是每句话的注意力权重可视化。从图中可以看到每句话中的每个单词占的比重不一样，所以编码重点输出的英文句子也有所差别。

经过注意力权重机制编码输出的翻译结果。

```
Out:
        input = elle a cinq ans de moins que moi .
        output = she is five years younger than me . <EOS>
        input = elle est trop petit .
        output = she s too hard . <EOS>
        input = je ne crains pas de mourir .
        output = i m not afraid . <EOS>
        input = c est un jeune directeur plein de talent .
        output = he s a lovely young of . <EOS>
```

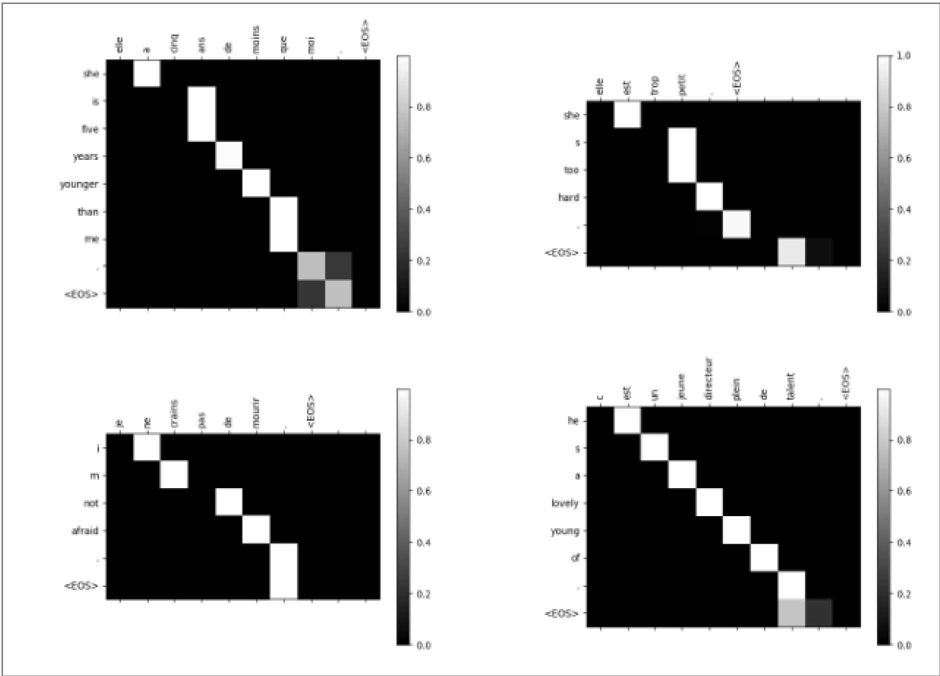


图 11.4　注意力权重可视化

第 12 章 利用 PyTorch 实现量化交易

随着计算机科学的发展，金融证券业经历了巨大的变革，从手工下单交易到由计算机下达指令完成指定任务。计算机的科学计算让复杂的统计过程变得简单，借助计算机科学实现各种统计指标的计算。由此借助现代统计学和数学的方法，利用计算机软件通过历史数据寻找并获得超额收益率，通过计算机程序，严格按照这些策略所构建的数量化模型进行投资，称为量化投资。量化投资利用模型对数据进行定性的思想量化，运用多种手段进行分析，从而选出能获得超额收益率的模型，形成量化策略。其中量化策略主要包括：量化择时、量化选股、统计套利、高频交易、股指期货套利、商品期货套利。量化投资有纪律性、系统性、套利思想、靠概率取胜的特点。利用计算机自动化下单克服了人性，如贪婪、恐惧的弱点，多层次多角度地对资产进行配置，包括行业选择、优选个股等。

在美国的金融行业中，西蒙斯管理的大奖章基金就是比较著名的量化基金，利用数学模型，实现程序化交易，所获得的收益率，在 1989 年到 2006 年之间的投资年化收益率平均高达 38.5%，远远超过巴菲特同期 21%的年化收益率。经历了几十年的发展，我国量化投资发展迅速，短短几年出现光大量化基金、中海量化基金、上投摩根阿尔法、沪深 300 增强等十几只量化投资基金。由于量化投资基金在我国出现的时间与美国量化投资发展的时间相比，还处于刚起步阶段，还有很大的发展空间。

股票市场从诞生到现在已经有四百多年的历史了，在西方国家已经发展得相当成熟了。各种理论应运而生，其中技术分析经历了一百多年的实践和总结，形成了一套完整的理论系统，根据各个指标的使用方式，广泛地应用到股票、期货、外汇等交易市场，已经成为投资分析中应用非常广泛的分析方法。从历史来看，技术分析主要建立在三个前提假设之上：市场的行为反应一切信息，价格呈趋势运动，历史会重演。技术指标是按照股票的开盘价、收盘价、最高价、最低价成交量，和成交金额和成交笔数进行数据的加工处理分析而得出的。以图表、技术指标为手段对证券市场的历史行为进行研究，运用一系列的方法进行归纳总结，从而达到预测证券市场价格变化趋势的目的。

由于股票的交易数据量巨大，因此深度学习算法模型就可以建立在大量的数据的基础上发现其规律。利用深度学习框架 PyTorch，采用深度学习算法模型来预测股价，就是一种不错的尝试。本章节包括利用 PyTorch 实现线性回归预测股价，前馈神经网络预测股价，递归神经网络预测股价等案例。在进行股票数据分析时，选择适合自己的量化投资模型，来进行股票交易。

12.1　线性回归预测股价

线性回归是利用数理统计中的回归分析，来确定两种或两种以上变量间相互依赖的定量关系的一种统计分析方法，运用十分广泛。在统计学中，线性回归（Linear Regression）是利用称为线性回归方程的最小平方函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析。这种函数是一个或多个被称为回归系数的模型参数的线性组合。只有一个自变量的情况称为简单回归，大于一个自变量的情况叫作多元回归。回归分析中，只包括一个自变量和一个因变量，且二者的关系可用一条直线近似表示，这种回归分析称为一元线性回归分析。

如果回归分析中包括两个或两个以上的自变量，且因变量和自变量之间是线性关系，则称为多元线性回归分析。线性回归模型经常用最小二乘来逼近来拟合。

前面讲了基本的线性回归以及模型框架，按照这个框架就可以写出一个线性回归模型。

首先我们看一下采集到的上证指数的数据，如图 12.1 所示。从 2017 年 1 月 3 日，包括开盘价、收盘价、最高价、最低价。涨跌采用 0 和 1 模式。如果当天的收盘价高于昨天的收盘价定义为 1，如果当天的收盘价低于昨天的收盘价定义为 0。

名称	代码	日期	开盘	最高	最低	收盘	涨跌
上证指数	980001	2017/1/3	3105.31	3136.46	3105.31	3135.92	1
上证指数	980001	2017/1/4	3133.79	3160.1	3130.11	3158.79	1
上证指数	980001	2017/1/5	3157.91	3168.5	3154.28	3165.41	1
上证指数	980001	2017/1/6	3163.78	3172.03	3153.03	3154.32	0
上证指数	980001	2017/1/9	3148.53	3173.14	3147.74	3171.24	1
上证指数	980001	2017/1/10	3167.57	3174.58	3157.33	3161.67	0
上证指数	980001	2017/1/11	3156.69	3167.03	3136.27	3136.75	0
上证指数	980001	2017/1/12	3133.6	3144.97	3115.98	3119.29	0
上证指数	980001	2017/1/13	3116.08	3130.51	3102.16	3112.76	0

图 12.1　证指数部分数据

同时采用前 100 天的开盘价、收盘价、最高价、最低价当作输入数据，第 2 天到 101 天的涨跌数据作为输出数据，来进行训练，进行股价次日涨跌预测。下面是具体的代码实现。

本例文件名为 PyTorch/Chapter12/pt1_linear_regression.py。

```
# -*- coding: utf-8 -*-
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torch.autograd as autograd
#导入模块库
df=pd.read_excel(r"/home/pc/桌面/上证指数数据.xlsx")
df1=df.iloc[:100,3:6].values
xtrain_features=torch.FloatTensor(df1)
df2=df.iloc[1:101,7].values
xtrain_labels=torch.FloatTensor(df2)
#用 pandas 库中的读取 excel 函数读取上证指数数据
```

```
xtrain=torch.unsqueeze(xtrain_features,dim=1)。
ytrain=torch.unsqueeze(xtrain_labels,dim=1)
x, y = torch.autograd.Variable(xtrain), Variable(ytrain)
class Net(torch.nn.Module):  # 继承 torch 的 Module
    def __init__(self, n_feature, n_hidden, n_output):
        super(Net, self).__init__()
        # 继承 __init__ 功能
        # 定义每层用什么样的形式
        self.hidden = torch.nn.Linear(n_feature, n_hidden)
        # 隐藏层线性输出
        self.predict = torch.nn.Linear(n_hidden, n_output)

    def forward(self, x):
        # 这同时也是 Module 中的 forward 功能
        # 正向传播输入值，神经网络分析出输出值
        x = F.relu(self.hidden(x))
        # 激励函数(隐藏层的线性值)
        x = self.predict(x)
        # 输出值
        return x
model = Net(n_feature=4, n_hidden=10, n_output=1)
```

需要定义 Loss 和 Optimizer，就是误差和优化函数

```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
```

这里使用的是最小二乘 Loss，之后我们做分类问题更多使用的是 Cross Entropy Loss，交叉熵。优化函数使用的是随机梯度下降，注意需要将 model 的参数 model.parameters()传进去让这个函数知道它要优化的参数是哪些。

```
num_epochs = 1000000
for epoch in range(num_epochs):
    inputs =x
    target =y
    out = model(inputs)
    # 前向传播
```

```
        loss = criterion(out, target)
        # 计算 loss
        optimizer.zero_grad()
        # 梯度归零
        loss.backward()
        # 反向传播
        optimizer.step()
        # 更新参数
        if (epoch+1) % 20 == 0:
            print('Epoch[{}/{}], loss:
{:.6f}'.format(epoch+1,num_epochs,loss.data[0]))
```

第一个循环表示每个 epoch，接着开始前向传播，然后计算 loss，然后反向传播，接着优化参数，特别注意的是在每次反向传播的时候需要将参数的梯度归零。

经过 100000 次训练之后的误差是 0.980399，说明经过大量训练之后，误差很小，说明精度还是不错的。

```
    Epoch[99800/100000], loss: 0.980399
    Epoch[99820/100000], loss: 0.980399
    Epoch[99840/100000], loss: 0.980399
    Epoch[99860/100000], loss: 0.980399
    Epoch[99880/100000], loss: 0.980399
    Epoch[99900/100000], loss: 0.980399
    Epoch[99920/100000], loss: 0.980399
    Epoch[99940/100000], loss: 0.980399
    Epoch[99960/100000], loss: 0.980399
    Epoch[99980/100000], loss: 0.980399
    Epoch[100000/100000], loss: 0.980399
```

训练完成之后我们就可以开始测试模型了。

```
    model.eval()
    predict = model(x)
    predict = predict.data.numpy()
    print(predict)
```

特别注意的是需要用 model.eval()，让 model 变成测试模式，这主要是因为 Dropout 和 Batch Normalization 的操作在训练和测试的时候是不一样的。

我们来看一下经过处理的前 10 个数据的预测结果：1，0，1，0，1，0，1，0，1。真实的数据如下：1，0，1，1，1，1，1，0，1。从中可以看出，经过 100000 次训练之后，由于数据量较少，可能出现过拟合现象，为此我们在进行预测的时候，需要进行参数的调整优化，找到最合适的参数进行训练。

12.2　前馈神经网络预测股价

前馈神经网络结构简单，应用广泛，能够以任意精度逼近任意连续函数及平方可积函数。而且可以精确实现任意有限训练样本集。从系统的观点看，前馈网络是一种静态非线性映射。通过简单非线性处理单元的复合映射，可获得复杂的非线性处理能力。从计算的观点看，缺乏丰富的动力学行为。大部分前馈网络都是学习网络，其分类能力和模式识别能力一般都强于反馈网络。由于影响股票价格数据的因素有许多，为此我们可以把这些因素当作输入来进行预测股价，为此下面两个案例分别进行股票数据的涨跌预测和股票数据的次日收盘价格进行预测。

首先，我们进行上证指数的次日收盘价价格进行预测。数据如图 12.2 所示。

名称	代码	日期	开盘	最高	最低	收盘
上证指数	980001	2017/1/3	3105.31	3136.46	3105.31	3135.92
上证指数	980001	2017/1/4	3133.79	3160.1	3130.11	3158.79
上证指数	980001	2017/1/5	3157.91	3168.5	3154.28	3165.41
上证指数	980001	2017/1/6	3163.78	3172.03	3153.03	3154.32
上证指数	980001	2017/1/9	3148.53	3173.14	3147.74	3171.24
上证指数	980001	2017/1/10	3167.57	3174.58	3157.33	3161.67
上证指数	980001	2017/1/11	3156.69	3167.03	3136.27	3136.75
上证指数	980001	2017/1/12	3133.6	3144.97	3115.98	3119.29

图 12.2　需要训练的部分数据

同时采用前 100 天的开盘价、收盘价、最高价、最低价当作输入数据，第 2 天到 101 天的收盘价数据作为输出数据，来进行训练，预测。下面是具体的代码实现。

案例名为 PyTorch/Chapter12/pt02_feedforward_neural_network.py

由于代码比较简单，我们重点分析数据结果。

```
import torch
import pandas as pd
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
df=pd.read_excel(r"/home/pc/上证指数数据.xlsx")

df1=df.iloc[:100,3:6].values
xtrain_features=torch.FloatTensor(df1)
df3=df.iloc[1:101,6].values
xtrain_labels=torch.FloatTensor(df2)
xtrain=torch.unsqueeze(xtrain_features,dim=1)
ytrain=torch.unsqueeze(xtrain_labels,dim=1)
x, y = torch.autograd.Variable(xtrain), Variable(ytrain)
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
net = Net(input_size=4, hidden_size=100, num_classes=1)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.005)
for epoch in range(100000):
    inputs =x
    target =y
    out =net(inputs)
    loss = criterion(out, target)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
        if (epoch+1) % 20 == 0:
            print('Epoch[{}], loss:
{:.6f}'.format(epoch+1,loss.data[0]))
```

我们可以看到，经过 100000 次训练之后，误差一直保持在 286 左右，误差已经很难缩小，为此，我们需要对参数进行优化，寻找更优参数，进行训练。

```
Epoch[99740], loss: 286.273773
Epoch[99760], loss: 286.270599
Epoch[99780], loss: 286.267883
Epoch[99800], loss: 286.265900
Epoch[99820], loss: 286.293884
Epoch[99840], loss: 539.946106
Epoch[99860], loss: 302.315033
Epoch[99880], loss: 291.796051
Epoch[99900], loss: 286.780792
Epoch[99920], loss: 286.455170
Epoch[99940], loss: 286.300873
Epoch[99960], loss: 286.276001
Epoch[99980], loss: 286.272491
Epoch[100000], loss: 286.27005
model.eval()
predict = model(x)
predict = predict.data.numpy()
print(predict)
```

我们来看一下经过处理的前 5 个数据的预测结果如下：

```
[[[ 3138.6640625 ]]
 [[ 3160.47827148]]
 [[ 3162.42553711]]
 [[ 3152.35766602]]
 [[ 3172.05029297]]
```

它与真实数据对比如下：

```
3158.79, 3165.41, 3154.32, 3171.24, 3161.67
```

经过对比，说明预测效果不太理想。

接下来，我们采用预测涨跌模式来对股价进行预测。

我们采用的数据如图 12.3 所示。

名称	代码	日期	开盘	最高	最低	收盘	涨跌
上证指数	980001	2017/1/3	3105.31	3136.46	3105.31	3135.92	1
上证指数	980001	2017/1/4	3133.79	3160.1	3130.11	3158.79	1
上证指数	980001	2017/1/5	3157.91	3168.5	3154.28	3165.41	1
上证指数	980001	2017/1/6	3163.78	3172.03	3153.03	3154.32	0
上证指数	980001	2017/1/9	3148.53	3173.14	3147.74	3171.24	1
上证指数	980001	2017/1/10	3167.57	3174.58	3157.33	3161.67	0
上证指数	980001	2017/1/11	3156.69	3167.03	3136.27	3136.75	0
上证指数	980001	2017/1/12	3133.6	3144.97	3115.98	3119.29	0
上证指数	980001	2017/1/13	3116.08	3130.51	3102.16	3112.76	0

图 12.3　部分上证指数数据

涨跌采用 0 和 1 模式。如果当天的收盘价高于昨天的收盘价定义为 1，如果当天的收盘价低于昨天的收盘价定义为 0。同时采用前 100 天的开盘价、收盘价、最高价、最低价当作输入数据，第 2 天到 101 天的涨跌数据作为输出数据，来进行训练，进行预测股价次日涨跌预测。下面是具体的代码实现。

案例名为 PyTorch/Chapter12/pt03_feedforward_neural_network03.py

```
import torch
import pandas as pd
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
df=pd.read_excel(r"/home/pc/上证指数数据.xlsx")

df1=df.iloc[:100,3:6].values
xtrain_features=torch.FloatTensor(df1)
df3=df["涨跌"].astype(float)
xtrain_labels=torch.FloatTensor(df2[:100])
xtrain=torch.unsqueeze(xtrain_features,dim=1)
ytrain=torch.unsqueeze(xtrain_labels,dim=1)
x, y = torch.autograd.Variable(xtrain), Variable(ytrain)
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
```

```
            super(Net, self).__init__()
            self.fc1 = nn.Linear(input_size, hidden_size)
            self.relu = nn.ReLU()
            self.fc2 = nn.Linear(hidden_size, num_classes)

        def forward(self, x):
            out = self.fc1(x)
            out = self.relu(out)
            out = self.fc2(out)
            return out
    net = Net(input_size=4, hidden_size=100, num_classes=1)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=0.005)
    for epoch in range(100000):
        inputs =x
        target =y
        out =net(inputs)
        loss = criterion(out, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if (epoch+1) % 20 == 0:
            print('Epoch[{}], loss:
{:.6f}'.format(epoch+1,loss.data[0]))
```

可以看到，经过 100000 次训练之后，误差逐渐缩小到 0.93 附近，说明经过多次训练之后，模型已经进行取得不错的效果了。

```
    Epoch[99700], loss: 0.932374
    Epoch[99720], loss: 0.930786
    Epoch[99740], loss: 0.930341
    Epoch[99760], loss: 0.930191
    Epoch[99780], loss: 0.930152
    Epoch[99800], loss: 0.930149
    Epoch[99820], loss: 0.985968
    Epoch[99840], loss: 0.933044
    Epoch[99860], loss: 0.931800
    Epoch[99880], loss: 0.930588
    Epoch[99900], loss: 0.930266
```

```
Epoch[99920], loss: 0.930168
Epoch[99940], loss: 0.930196
Epoch[99960], loss: 0.951381
Epoch[99980], loss: 0.934654
Epoch[100000], loss: 0.931125
```

12.3 递归神经网络预测股价

股票市场是一个复杂的非线性动力学系统，恰好递归神经网络具有非线性映射能力，自学习能力的优点，非常适合于股票的相关性分析和预测。股票的预测是一个动态的时间建模问题，我们在预测个股的第二天涨跌情况，或者是第二天的收盘价格，只是利用过去的历史数据的开盘价、收盘价、最高价、最低价等进行建模分析。这种问题相当于一个时间序列问题，递归神经网络恰恰适合处理这种问题。

递归神经网络（RNN）是两种人工神经网络的总称。一种是时间递归神经网络（Recurrent Neural Network），另一种是结构递归神经网络（Recursive Neural Network）。时间递归神经网络的神经元间连接构成有向图，而结构递归神经网络利用相似的神经网络结构递归构造更为复杂的深度网络。两者训练的算法不同，但属于同一算法变体。

传统的神经网络模型中，我们假设所有的输入是互相独立的。从输入层到隐藏层再到输出层，层与层之间是全连接的，每层之间的节点是无连接的。循环神经网络与传统的前馈神经网络有所不同，循环神经网络的隐藏层相互连接，即一个序列当前的输出与前面的输出也有关。

循环神经网络会对每一个时刻的输入结合当前模型的状态给出一个输出。RNN 是对序列的每个元素执行同样的操作，其输出依赖于前次计算的结果。RNN 引入了定向循环，能够处理那些输入之间前后关联的问题。RNN 拥有捕获已计算节点信息的记忆能力。

LSTM 网络预测股价分类和回归

LSTM 长短时模型是最适合金融数据、量化分析的神经网络模型之一，既可以用于数值预测，也可以用于趋势预测。LSTM 算法，具有 RNN 循环

神经网络算法的各种优点，但因为 LSTM 模型有更好的记忆能力，所以效果更佳。利用深度神经网络中的 LSTM 的一些基本结果，对多因子模型进行尝试，以检验深度神经网络在多因子、投资领域的适用性，使得投资者能够对神经网络有更为实际的理解，并能够在投资领域有所运用。

下文，我们使用 LSTM 模型分别进行第二天股价收盘价的预测与第二天股价涨跌的预测。

首先，我们进行上证指数的次日收盘价价格进行预测。数据如图 12.4 所示。

名称	代码	日期	开盘	最高	最低	收盘
上证指数	980001	2017/1/3	3105.31	3136.46	3105.31	3135.92
上证指数	980001	2017/1/4	3133.79	3160.1	3130.11	3158.79
上证指数	980001	2017/1/5	3157.91	3168.5	3154.28	3165.41
上证指数	980001	2017/1/6	3163.78	3172.03	3153.03	3154.32
上证指数	980001	2017/1/9	3148.53	3173.14	3147.74	3171.24
上证指数	980001	2017/1/10	3167.57	3174.58	3157.33	3161.67
上证指数	980001	2017/1/11	3156.69	3167.03	3136.27	3136.75
上证指数	980001	2017/1/12	3133.6	3144.97	3115.98	3119.29

图 12.4　需要训练的部分数据

我们采用前 100 天的开盘价、收盘价、最高价、最低价数据当作输入数据，第二天到 102 天的收盘价当作输出数据进行训练。

案例名为 PyTorch/Chapter13/pt04_rnn.py。

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
import pandas as pd
#加载所需的模块包

#设置参数

input_size = 1
hidden_size = 100
```

```
    num_layers = 10
    num_classes = 1

    df=pd.read_excel(r"/home/pc/上证指数数据.xlsx")
    df1=df.iloc[:100,3:6].values
    xtrain_features=torch.FloatTensor(df1)
    df3=df.iloc[1:101,6].values
    xtrain_labels=torch.FloatTensor(df2)

    xtrain=torch.unsqueeze(xtrain_features,dim=1)

    ytrain=torch.unsqueeze(xtrain_labels,dim=0)
    x1=torch.autograd.Variable(xtrain_features.view(100,4,1))
    x, y = torch.autograd.Variable(xtrain), Variable(ytrain)
    #定义循环神经网络结构
    class RNN(nn.Module):
        def __init__(self, input_size, hidden_size, num_layers,
num_classes):
            super(RNN, self).__init__()
            self.hidden_size = hidden_size
            self.num_layers = num_layers
            self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
            self.fc = nn.Linear(hidden_size, num_classes)

        def forward(self, x):
            h0 = Variable(torch.zeros(self.num_layers, x.size(0),
self.hidden_size))
            c0 = Variable(torch.zeros(self.num_layers, x.size(0),
self.hidden_size))

            out, _ = self.lstm(x, (h0, c0))
            out = self.fc(out[:, -1, :])
            return out

    rnn = RNN(input_size, hidden_size, num_layers, num_classes)

    #损失函数以及优化函数
```

```
#训练模型
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=0.005)
for epoch in range(100000):
    inputs =x1
    target =y
    out =rnn(inputs) # 前向传播
    loss = criterion(out, target) # 计算 loss
    # backward
    optimizer.zero_grad() # 梯度归零
    loss.backward() # 方向传播
    optimizer.step() # 更新参数

    if (epoch+1) % 20 == 0:
        print('Epoch[{}], loss:
{:.6f}'.format(epoch+1,loss.data[0]))
```

我们可以看到，经过 100000 次训练之后，误差大小为 711，说明经过大量训练，误差仍不能缩小，可能是模型的参数需要进一步优化。在训练神经网络过程中，“过拟合”是一项尽量要避免的事。神经网络“死记”训练数据。过拟合意味着模型在训练数据的表现会很好，但对于训练以外的预测则效果很差。原因通常为模型“死记”训练数据及其噪声，从而导致模型过于复杂。本文使用的上证指数的数据量不是太多，因此防止模型过拟合就尤为重要。

```
Epoch[99900], loss: 736.000000
Epoch[99920], loss: 731.500000
Epoch[99940], loss: 726.500000
Epoch[99960], loss: 721.500000
Epoch[99980], loss: 716.500000
Epoch[100000], loss: 711.000000
```

接下来，我们来看一下模型预测的结果

```
model.eval()
predict = model(x)
predict = predict.data.numpy()
```

```
    print(predict)
```

我们来看一下经过处理过的前 10 个数据的预测结果如下：1，0，0，0，0，0，1，0，1。真实的数据如下：1，0，1，1，1，1，1，0，1。从中可以看出，经过 100000 次训练之后，由于数据量较少，可能出现过拟合现象，为此我们在进行预测的时候，需要进行参数的调整优化，找到最合适的参数进行训练。

训练 LSTM 模型时，在参数层面上有两个十分重要的参数可以控制模型的过拟合：Dropout 参数和在权重上施加正则项。Dropout 是指在每次输入时随机丢弃一些 Features，从而提高模型的鲁棒性。它的出发点是通过不停去改变网络的结构，使神经网络记住的不是训练数据本身，而是能学出一些规律性的东西。正则项则是通过在计算损失函数时增加一项 L2 范数，使一些权重的值趋近于 0，避免模型对每个 Feature 强行适应与拟合，从而提高鲁棒性，也有因子选择的效果。

接下来，我们采用预测涨跌模式来对股价进行预测。

我们采用的数据如图 12.5 所示。

名称	代码	日期	开盘	最高	最低	收盘	涨跌
上证指数	980001	2017/1/3	3105.31	3136.46	3105.31	3135.92	1
上证指数	980001	2017/1/4	3133.79	3160.1	3130.11	3158.79	1
上证指数	980001	2017/1/5	3157.91	3168.5	3154.28	3165.41	1
上证指数	980001	2017/1/6	3163.78	3172.03	3153.03	3154.32	0
上证指数	980001	2017/1/9	3148.53	3173.14	3147.74	3171.24	1
上证指数	980001	2017/1/10	3167.57	3174.58	3157.33	3161.67	0
上证指数	980001	2017/1/11	3156.69	3167.03	3136.27	3136.75	0
上证指数	980001	2017/1/12	3133.6	3144.97	3115.98	3119.29	0
上证指数	980001	2017/1/13	3116.08	3130.51	3102.16	3112.76	0

图 12.5　部分上证指数数据

涨跌采用 0 和 1 模式。如果当天的收盘价高于昨天的收盘价定义为 1，如果当天的收盘价低于昨天的收盘价定义为 0。采用开盘价、收盘价、最高价、最低价当作输入数据，涨跌作为输出数据，来进行训练，进行预测股价次日涨跌预测。实现代码如下：

案例名为 PyTorch/Chapter13/pt05_rnn.py。

```
    import torch
```

```
    import torch.nn as nn
    import torchvision.datasets as dsets
    import torchvision.transforms as transforms
    from torch.autograd import Variable
    import pandas as pd
    #加载所需的模块包

    #设置参数

    input_size = 1
    hidden_size = 100
    num_layers = 10
    num_classes = 1

    df=pd.read_excel(r"/home/pc/上证指数数据.xlsx")

    df1=df.iloc[:100,3:6].values
    xtrain_features=torch.FloatTensor(df1)
    df3=df["涨跌"].astype(float)
    xtrain_labels=torch.FloatTensor(df2[:100])



    xtrain=torch.unsqueeze(xtrain_features,dim=1)

    ytrain=torch.unsqueeze(xtrain_labels,dim=0)
    x1=torch.autograd.Variable(xtrain_features.view(100,4,1))
    y = torch.autograd.Variable(ytrain)
    #定义循环神经网络结构
    class RNN(nn.Module):
        def __init__(self, input_size, hidden_size, num_layers,
num_classes):
            super(RNN, self).__init__()
            self.hidden_size = hidden_size
            self.num_layers = num_layers
            self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
            self.fc = nn.Linear(hidden_size, num_classes)
```

```
        def forward(self, x):
            h0 = Variable(torch.zeros(self.num_layers, x.size(0),
self.hidden_size))
            c0 = Variable(torch.zeros(self.num_layers, x.size(0),
self.hidden_size))


            out, _ = self.lstm(x, (h0, c0))
            out = self.fc(out[:, -1, :])
            return out

    rnn = RNN(input_size, hidden_size, num_layers, num_classes)

    #损失函数以及优化函数


    #训练模型
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(rnn.parameters(), lr=0.005)
    for epoch in range(10000):
        inputs =x1
        target =y
        out =rnn(inputs) # 前向传播
        loss = criterion(out, target) # 计算 loss
        # backward
        optimizer.zero_grad() # 梯度归零
        loss.backward() # 方向传播
        optimizer.step() # 更新参数

        if (epoch+1) % 20 == 0:
            print('Epoch[{}], loss:
{:.6f}'.format(epoch+1,loss.data[0]))
```

我们可以看到，经过 100000 次训练之后，误差一直保持在 0.98 左右，误差已经很难缩小，觉得模型效果还是不错的，为此我们进行下一步的预测。

```
    Epoch[99920], loss: 0.980637
    Epoch[99940], loss: 0.980424
```

```
Epoch[99960], loss: 0.980403
Epoch[99980], loss: 0.980401
Epoch[100000], loss: 0.980401
model.eval()
predict = model(x)
predict = predict.data.numpy()
print(predict)
```

我们来看一下经过处理的前 5 个数据的预测结果如下（保留两位有效数字）：

```
[[[ 3150.66]]
 [[ 3140.47]]
 [[ 3172.42]]
 [[ 3162.35]]
 [[ 3152.05]] ]
```

它与真实数据对比如下：

```
3158.79，3165.41，3154.32，3171.24，3161.67
```

经过对比，说明预测效果不太理想。

利用基本的 LSTM 结构，未能够得到高的准确率与显著水平，对于模型的进一步改进和优化令人有所期待。同时 LSTM 强大的数据处理能力必将在投资领域展露出来，也同样令人期待。